



S2 Labs



SALESFORCE HULK

LEARN SALESFORCE DEVELOPMENT WITH S2 LABS

**COURSE DEVELOPED
AND CURATED BY
-SHREY SHARMA**

WWW.SHREYSHARMA.COM

COPYRIGHT

© COPYRIGHT | S2 LABS ALL RIGHTS RESERVED.

VARIOUS TRADE MARK SHIELD BY RESPECTIVE OWNERS.

THIS DOCUMENT CONTAINS PROPRIETARY AND CONFIDENTIAL INFORMATION OF S2 LABS,THIS E-BOOK OR ANY PORTION THEREOF MAY NOT BE REPRODUCED OR USED IN ANY MANNER WHATSOEVER, EXCEPT FOR EDUCATIONAL AND INFORMATIONAL PURPOSES.

Table Of Contents

Topics	Page No.
Chapter 1 - Apex Basics.....	01
1.1. What is Apex and its Features	
1.2. When to Use and Flows of Action	
1.3. Features not supported by APEX	
1.4. What are Apex Environments?	
1.5. Tools for Writing Code	
1.6. Apex Variables and Apex Literals	
Chapter 2 - Datatypes and Operators.....	05
2.1 APEX Data Types	
2.2 sObjects	
2.3 Generic sObjects	
2.4 Enums	
2.5 Rules of Conversion	
2.6 Collections	
2.7 Lists	
2.8 Nested Lists	
2.9 Sets	
2.10 Maps	

2.11 Operators

Chapter 3 - Logic Control and Looping Statements.....19

3.1 Logic Control and Looping Statements

3.2 Looping Assignment Questions

Chapter 4 - SOQL.....23

4.1 SOQL Basics

4.2 How to Write SOQL in APEX

4.3 SOQL Variable Binding

4.4 SOQL Return Type

4.5 SOQL Functions

4.6 SOQL Keywords

4.7 Date Literals

4.8 Child to Parent Relationship

4.9 Parent to Child Relationship

4.10 Multi-level Relationships

4.11 Dynamic SOQL

Chapter 5 - SOSL.....32

5.1 What is SOSL?

5.2 How to Access Record in Apex Returned by SOSL

5.3 Return Specific Fields

5.4 Wildcards in SOSL

5.5 SOQL v/s SOSL

Chapter 6 - Apex DML Statements and Database Method.....37

6.1 Apex DML Basics

6.2 DML SOQL Best Practices

6.3 Database Class

6.4 Empty Recycle Bin

6.5 Count Query

6.6 Lead Conversion

6.7 Transaction Control and Rollback

6.8 Database Class Method Result Object

Chapter 7 - Exceptional Handling.....47

7.1 Basics

7.2 System Defined Exceptions and Types

7.3 Custom or User Defined Exceptions

Chapter 8 - ApexTriggers.....57

8.1 Basics

8.2 Definition

8.3 Types of Trigger

8.4 Trigger Order of Execution

- 8.5 Trigger Context Variables
- 8.6 Best Practices for Triggers
- 8.7 Best Practice Part 2
- 8.8 Trigger Exceptions

Chapter 9 - Apex Testing.....75

- 9.1 Basics
- 9.2 System-Defined Methods
- 9.3 Test Data
- 9.4 Unit Tests

Chapter 10 - Governor Limits and Batch Apex.....87

- 10.1 What are Governor Limits?
- 10.2 Governor Limits Types
- 10.3 Batch Class
- 10.4 Schedule Batch Class in APEX
- 10.5 Execute Batch Class in APEX
- 10.6 Access Modifiers
- 10.7 Namespace

Chapter 11 - Visualforce.....96

- 11.1 Basics**
- 11.2 Architecture

11.3 Standard Controller

11.4 View Pages

11.5 Edit Pages

11.6 Custom Controller

11.7 Order of Execution

11.8 Tags

11.9 System Mode



S2 Labs



SALESFORCE HULK

CHAPTER - 1

APEX BASICS



SALESFORCE DEVELOPMENT COURSE



WHAT IS APEX?

- Apex is a strongly typed, object-oriented programming language which is a proprietary language developed by salesforce.com to allow us to write the code that executes on the force.com platform.
- It is used for building SAAS applications on top of salesforce.com CRM functionality.
- Apex is saved, compiled and executed on the servers of the force.com platform.
- It enables the developer to add business logic to most of the system events including button on-clicks, related record updates and VF & lightning pages.

FEATURES OF APEX:

- ★ It upgrades automatically.
- ★ Integrated with the DB which means it can access and manipulate records without the need to establish the DB connection explicitly.
- ★ It has Java like syntax and it is easy to use.
- ★ It is easy to test as it provides built-in support for executing test cases.
- ★ Multi-Tenant Environment.
- ★ You can save your apex code against different versions of the force.com API.
- ★ Apex is a case-insensitive language.

WHEN TO USE APEX?

1. To perform a couple of business processes which are not supported by workflows or processes or flows.
2. To perform complex validation over multiple objects.
3. To create web services and email services.
4. For transactions and rollbacks.

FLOWS OF ACTION:

- A. Developer Action: When the developer writes and saves the code to apex platform. The platform application server compiles the code into a set of instruction that can be understood by the apex-runtime, interprets and then saves those instructions as compiled apex.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

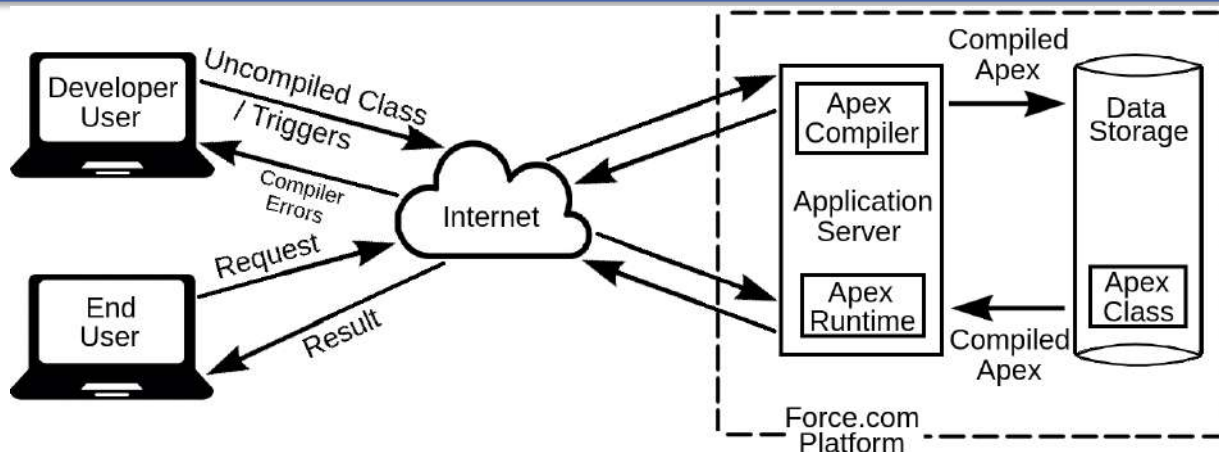
training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



- B. End-User Action: When the end-user performs some action which involves the apex code or when the end-user triggers the execution of apex by clicking a button or accessing a VF page. The platform application server retrieves the compiled instructions from the meta-data and sends it through apex runtime which interprets before returning the result.

The end-user observes no differences in the execution time as compared to the standard application platform request.

FEATURES THAT ARE NOT SUPPORTED BY APEX:

1. It cannot show the elements in UI other than error message.
2. It cannot change the standard SFDC functionality but can be used to stop its execution or to add a new functionality.
3. It cannot be used to create a temporary file.
4. It cannot create multiple threads.

APEX ENVIRONMENTS:

There are several environments for developing apex code

- We can run apex in a developer org, production org and a sandbox.
- We cannot develop in our production org because live users are accessing the system so while we are developing it can destabilize our data and corrupt our application.
- Instead we do all our development work in sandbox or developer edition.

DIFFERENT TOOLS FOR WRITING APEX CODE:

1. Force.com Developer Console: The developer console is an integrated developer environment and collection of tools that we can use to create, debug and test applications in our salesforce org.
2. Code Editor in Salesforce Interface: This code editor compiles all classes and triggers then they are some and flags the errors if there are any.



SALESFORCE DEVELOPMENT COURSE



Note: The Code doesn't get saved until it compiles without errors. The force.com developer console allows you to write, test and debug our Apex code. On the other hand the code editor in the user interface enables only writing the code and does not support debugging or testing.

APEX VARIABLES:

A variable is a named value holder in memory. In Apex, local variables are declared with Java-like syntax.

The name we choose for a variable is called an identifier. Identifier can be of any length but it must begin with an alphabet. The rest of the identifier can include digits also.

- Operators and spaces are not allowed.
- We can't use any of the apex reserved keywords when naming variables, methods or classes.
- Keywords are the words that are essential for Apex language.

```
Integer i_a; ❌
```

```
Integer _ia; X
```

```
Integer ia_; X
```

```
Integer i2; ❌
```

```
Integer i 2; X
```

```
Integer i$2; X
```

```
Integer $i2; X
```

```
Integer $_i_2; X
```

APEX CONSTANTS:

Apex Constants are the variables whose values don't change after being initialized once.

Constants can be defined using the final keyword.

Constants can be assigned almost once either in the declaration itself or with a static initialization method, if the constant is defined in a class.

```
final integer a =5;
```

Note: We cannot use the final keyword in the declaration of a class or a method because in apex classes and methods are by-default final and cannot be overridden or inherited without using the virtual keyword.

APEX LITERALS:

```
Integer i = 123;  
String str = 'abc';
```

Expression:

An Expression is a combination made up of variables, operators and method invocation that evaluates to a single value.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs



SALESFORCE HULK

CHAPTER - 2

APEX DATATYPES AND OPERATORS

DATATYPES:

Apex is a strongly typed language i.e we must declare the datatype of a variable when we first refer it.

All apex variables are initialized to null by-default.

Primitive Datatypes:

1. Integer Types:

There are 2 datatypes that we can use to store values.

- Integer (4 bytes) → Range(-2147483647 to +2147483647)
- Long (8 bytes) → Range (-2^{63} to $+2^{63}-1$)

Note: If the literal goes out of integer range then append l or L at the end of the literal because by-default a number is treated as integer. Eg. long L = 3434343434L;

Note: If the value is not in the range of type defined then the compiles time error is generated.

2. Floating Point Datatypes:

A floating point variable can represent a very wide range but with a fix number of digits in accuracy.

- Double (8 bytes)
- Decimal: A number that includes a decimal part. Decimal is a arbitrary precision number.

Note: Currency fields are automatically define as type decimal.

3. Date, Time and DateTime:

A value that indicates a particular date and time.

Ex: Date d = Date.newInstance(2016,6,15);

A value that indicates a particular time. Time values must always be created with a system static method. Time datatype stores time (hours, minutes, seconds, milliseconds).

Ex: Time t = Time.newInstance(12,5,2,7);

A value that indicates a particular date and time.

Ex: DateTime dt = Date.newInstance(1997,1,31,7,8,16);

4. Boolean:

This variable can either be true or false or null.

5. String:

Any set of characters surrounded by single quotes. String can be null or empty and can include leading and trailing spaces.

LI

Ex: String s = 'a';

6. ID:

Any valid ID (18 character force.com identifier). If you set ID to a 15 character value then apex converts its value to its 18 character representation.

Note: All invalid ID values are rejected with a runtime exception. A collection of binary data stored as a single object.

7. BLOB:

Blob is typically used to store images, audio or other multimedia objects and sometimes binary executable code is also stored as a blob. We can convert this datatype to string or from string using the `toString()` and `valueOf()` methods.

Ex:

```
String s = 'abc';
Blob b = Blob.valueOf(s);
String s1 = b.toString();
```

sObjects and Generic sObjects:

Unlike any other programming language like Java or C#, Apex is tightly integrated with the database.

Hence we do not have to create any database connection to access the records or insert new records.

Instead, in Apex, we have sObjects which represent a record in Salesforce.

For example:

An account record named as Burlington Textiles in apex will be referred using an sObject, like this:

```
Account acc = new Account(Name='Disney');
```

The API object name becomes the data type of the sObject variable in Apex.

Here,

Account	=	sObject datatype
acc	=	sObject variable
new	=	keyword to create new sObject Instance
Account()	=	Constructor which creates an sObject instance
Name = 'Disney'	=	Initializes the value of the Name field in account sObject

Similarly if we want to create a contact record using Apex then we first need to create a sObject for it in Apex, like this:

```
Contact con = new Contact();
```

Now there are 2 ways to assign field values to the contact sObject:

1. **Through Constructor:**

```
Contact con = new Contact(firstName = 'Shrey', lastName = 'Sharma');
```

2. **Using dot notation:**

```
Contact con = new Contact();  
con.firstName = 'Shrey';  
con.lastName = 'Sharma';
```

Similarly for Custom Objects:

```
Student__c st = new Student__c(Name = 'Arnold');
```

If we want to assign the field values for custom fields then also we have to write down their field API name, like:

For standard object:

```
Account acc = new Account(Name = 'Disney', NumberOfLocations__c = 56);
```

For custom object:

```
Student__c st = new Student__c(Name = 'Arnold', Email__c = 'arnold@gmail.com');
```

Generic sObject:

Generally while programming we use specific sObject type when we are sure of the instance of the sObject but whenever there comes a situation when we can get instance of any sObject type, we use generic sObject.

Generic sObject datatype is used to declare the variables which can store any type of sObject instance.

Ex

```
sObject s1 = new Account(Name = 'Disney');
```

```
sObject s2 = new Contact(lastName = 'Sharma');
```

```
sObject s3 = new Student__c(Name = 'Arnold');
```

sObject Variable ---> Any sObject Datatype instance

Note: Every Salesforce record is represented as a sObject before it gets inserted into the database and also if you retrieve the records already present into the database they are stored in sObject variable.

Note: We can also cast the generic sObjects in specific sObjects like this:

```
Account acc = (Account) s1;
```

```
Contact con = (Contact) s1; //Will throw a Runtime Exception: datatype mismatch
```

1. Setting and Accessing values from Generic sObjects:

Similar to the sObject, we can also set and access values from Generic sObject. However, the notation is a little different.

Examples of setting the values and accessing them:

- a) Set a field value on an sObject


```
sObject s = new Account();
s.put('Name', 'Cyntexa Labs');
```
- b) Access a field on an sObject


```
Object objValue = s.get('Name');
```

Enums:

An enum is an abstract datatype with values that each take on exactly one of the finite set of identifiers that you specify. Enums are typically used to define a set of possible values that don't otherwise have a numerical order such as suit of card or a particular season of the year.

Ex:

```
Public enum season(winter, summer, spring, fall);
Season s=season.winter; // define it into a new file to make it global
System.debug(s);
```

Rules of conversion:

In general, apex requires explicit conversion from one datatype to another however a few datatype can be implicitly converted like variables of lower numeric type to higher types.

Hierarchy of numbers: integer<long<double<decimal

Note: Once a value has been passed from a number of lower type to a number of higher type, the value is converted to the higher type of number.

- Ids can always be assigned to strings.
- String can be assigned to ids however at runtime the value is checked to ensure that it is a legitimate id, if it is not then a runtime error is thrown.

- If the numeric value of the right hand side exceeds the maximum value for an integer you get a compilation error. In this case, the solution is to append L or l to the numeric value so that it represents a long value which has a wider range.
- Arithmetic computation that produce values larger than the max values of the current type are set to overflow.
Ex: Integer i = 2147483647 + 1; // overflow

Collections

Apex language provides developer with three classes (Set, List, Map) that makes it easier to handle collection of objects. In a sense, these collections works somewhat like arrays except their size can change dynamically and they have more advanced behaviours and easier access methods than arrays.

1. List :

Violet	Indigo	Blue	Green	Yellow	Orange	Red
0	1	2	3	4	5	6

Lists are used to store data in sequence. These are the widely used collections which can store primitives, user defined objects, sObjects, Apex Objects or other collection.

There are 2 main properties of lists:

- It stores data in sequential order.
- The data which it stores is non-unique or can be duplicate.

Hence use a list when the sequence of elements is important and where uniqueness of the elements are not important.

List initialization

```
List<datatype> list1 = new List<datatype>();
```

Example:

```
List<String> stList = new List<String>();
stList.add('ABC');
stList.add('DEF');
stList.add('FGH');
```

OR

```
List<String> stList = new List<String>{'ABC', 'DEF', 'FGH'};
```

List Array Notation

Developers comfortable with Array notation can also use it while using Lists. Array and List syntax can be used interchangeably. Array notation is a notation where a list's elements are referenced by enclosing the index number in square brackets after the list's name.


```
String[] nameList = new String[4];
//or
String[] nameList2 = new List<String>();
//or
List<String> nameList3 = new String[4];
// Set values for 0, 1, 2 indices
nameList[0] = 'Bhavna';
nameList[1] = 'Bhavya';
nameList[2] = 'Swati';
// Size of the list, which is always an integer value
Integer listSize = nameList.size();
// Accessing the 2nd element in the list, denoted by index 1
System.debug(nameList[1]); // 'Bhavna'
```


Some common methods of list:

add(element)	It adds an element into the list. it always adds at the last.	List<String> l = new List<String>(); l.add('abc'); System.debug(l); // ('abc')
size()	Returns the number of elements in the list.	l.add('def'); System.debug(l.size()); // 2 System.debug(l); // ('abc', 'def')
get(index)	Returns the element on the ith index.	System.debug(l.get(0)); // 'abc'
remove(index)	Removes the element on ith index.	l.remove(1); System.debug(l); // ('abc')
clone()	Makes a duplicate of a list.	List<String> l2 = l.clone();
set()	Sets the element on the ith position of the list. If there is already a value then value gets overridden.	l.add('def'); l.add('ghi'); System.debug(l); // ('abc', 'def', 'ghi') l.set(2, 'aaa'); System.debug(l); // ('abc', 'def', 'aaa')
sort()	Sorts the item in ascending order but works with primitive datatypes	l.sort(); System.debug(l); // ('aaa', 'abc', 'def')

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

	only.	
isEmpty()	Returns true if the list is empty.	<code>System.debug(l.isEmpty()); // false</code>
clear()	Clears the list.	<code>l.clear()</code>

Lists can be multidimensional also:

- 2-Dimensional:
`List<List<Integer>> nestList = new List<List<Integer>>();`
- 3-Dimensional:
`List<List<List<Integer>>> nestList = new List<List<List<Integer>>>();`

Note: SOQL queries also returns list of records.

2. Set :

Value	'Lists'	'Sets'	'Maps'
-------	---------	--------	--------

A set is a collection of unique, unordered elements.

It can contain primitive datatypes (string, integer, date, etc.) , sObjects or user-defined objects.

The basic syntax of creating a set :

`Set<datatype> set1 = new Set<datatype>;` OR
`Set<datatype> = new Set<datatype>{value1, value2,...};`

- The main characteristic of a set is the uniqueness of the elements. You can safely try to add the same element to a set more than once and it will disregard the duplicate without throwing an exception.

`Set<String> collections = new Set<String>{'Lists', 'Sets', 'Maps'};`
`s1.add('lists');`

- However, there is a slight twist with sObject i.e. the uniqueness of the data respects Case Sensitivity.

- Uniqueness of sObject is determined by comparing fields in their object.
- The way you declare sets is similar to the way you declare lists. You can also have nested sets and sets of lists.

Ex: `// Empty set initialized`
`Set<String> strSet = new Set<String>();`
`// Set of List of Strings`
`Set<List<String>> set2 = new Set<List<String>>();`

- Some common methods of set:

add(element)	Adds an element to set and only takes the	<code>Set<String> s = new Set<String>();</code> <code>s.add('abc');</code>
--------------	---	---

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com)

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



	argument of special datatype while declaring the set.	<pre>s.add('ABC'); s.add('abc'); System.debug(s); // ('abc', 'ABC')</pre>
addAll(list/set)	Adds all of the elements in the specified list/set to the set if they are not already present.	<pre>List<String> l = new List<String>(); l.add('abc'); l.add('def'); s.addAll(l); System.debug(s); // ('abc', 'ABC', 'def')</pre>
clear()	Removes all the elements.	<pre>s.clear(); s.addAll(l);</pre>
clone()	Makes duplicate of a set.	<pre>List<String> s2 = s.clone(); System.debug(s); // ('abc', 'def')</pre>
contains(elm)	Returns true if the set contains the specified element.	<pre>Boolean result = s.contains('abc'); System.debug(result); // true</pre>
containsAll(list)	Returns true if the set contains all of the elements in the specified list. The list must be of the same type as the set that calls the method.	<pre>Boolean result = s.containsAll(l); System.debug(result); // true</pre>
size()	Returns the size of set.	<pre>System.debug(s.size()); // 2</pre>
retainAll(list)	Retains only the elements in this set that are contained in the specified list and removes all other elements.	<pre>s.add('ghi'); System.debug(s); // ('abc', 'def', 'ghi') s.retainAll(l); System.debug(s); // ('abc', 'def')</pre>
remove(elm)	Removes the specified element from the set if it is present.	<pre>s.add('ghi'); s.remove('ghi'); System.debug(s); // ('abc', 'def')</pre>
removeAll(list)	Removes the elements in the specified list from the set if they are present.	<pre>s.add('ghi'); s.removeAll(l); System.debug(s); // ('ghi')</pre>

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com)

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

3. Map :

130	131	132	133	134	135
'Bhavya'	'Divya'	'Bhawna'	'Sapna'	'Pushpa'	'Harsha'

a) Map is collection of key, value pairs. Keys can be any primitive datatype while values can include primitives, apex objects, sObjects and other collections.

b) Use maps when want to quickly find out something with help of a key, each key must be unique but you can have duplicate values in your map.

c) Syntax for declaration:

```
Map<Datatype_Key, Datatype_value> m = new Map<Datatype_Key,
Datatype_value>();
```

d) To create a map of all the account records in your org, the Salesforce ID should be the key, which is the unique identifier for all the values of the account record.

```
Ex: // Nested map
Map<ID, Set<String>> m = new Map<ID, Set<String>>();
```

e) Initialisation of Map:

```
Map<Integer, String> m = new Map<Integer, String>{5=>'Kalpana'};
m.put(1, 'Bhavna');
m.put(2, 'Sapna');
m.put(3, 'Divya');
m.put(4, 'Bhavya');
m.put(2, 'Divya'); // this will override previous value
```

f) Map of List:

```
Map<Integer, List<Integer>> m = new Map<Integer, List<Integer>>();
m.put(1, new List());
```

g) Methods of Map:

put(key,value)	Associates the specified value with the specified key in the map.	<pre>Map<Integer,String> m = new Map<Integer,String>(); m.put(1, 'Bhavna'); m.put(2, 'Sapna'); System.debug(s); // (1='Bhavna', 2='Sapna')</pre>
putAll(map)	Copies all of the mappings from the specified map to the original map.	<pre>Map<Integer,String> n = new Map<Integer,String>(); n.put(3, 'Bhavna'); n.put(4, 'Divya'); m.putAll(n); System.debug(m); // (1='Bhavna', 2='Sapna' 3='Bhavna', 4='Divya')</pre>
get(key)	Returns the value to which	<pre>System.debug(m.get(1)); // 'Bhavna' System.debug(m.get(2)); // 'Sapna' System.debug(m.get(3)); // 'Bhavna'</pre>

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



	the specified key is mapped, or null if the map contains no value for this key.	<code>System.debug(m.get(4)); // 'Divya'</code>
<code>values()</code>	Returns a list that contains all the values in the map.	<code>List<String> l = new List<String>(); l = m.values(); System.debug(l); // ('Bhavna','Sapna','Bhavya','Divya')</code>
<code>clone()</code>	Makes a duplicate copy of the map.	<code>Map<Integer,String> o = new Map<Integer,String>(); o = m.clone(); System.debug(o); // (1='Bhavna',2='Sapna' 3='Bhavya',4='Divya')</code>
<code>keySet()</code>	Returns a set that contains all of the keys in the map.	<code>System.debug(m.keySet()); // (1,2,3,4)</code>
<code>containsKey(key)</code>	Returns true if the map contains a mapping for the specified key.	<code>Boolean result1 = m.containsKey(3); System.debug(result1); // true Boolean result2 = m.containsKey(6); System.debug(result2); // false</code>
<code>isEmpty()</code>	Returns true if map has 0 key.	<code>Boolean result = m.isEmpty(); System.debug(result); // false</code>
<code>size()</code>	Returns the number of key-value pairs in the map.	<code>System.debug(m.size()); // 4</code>
<code>remove(key)</code>	Removes the mapping for the specified key from the map, if present, and returns the corresponding value.	<code>m.remove(4); System.debug(m); // (1='Bhavna',2='Sapna' 3='Bhavya')</code>

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

clear()	Removes all of the key-value mappings from the map.	m.clear();
---------	---	------------

Note: Maps are used frequently to store records that you want to process or as containers for lookup data. It is very common to query for records and store them in a map so that you can do something with them.

Operators:

1. Add/Sub:

This operator adds or subtracts the value of Y from the value of X according to the following rules.

- If X and Y are integers or doubles then adds or subtracts the value of Y from X and if any X or Y is double then the result is double.
- If X is a date and Y is an integer then it returns a new date that is incremented or decremented by the specified number of days.
- If X is a DateTime and Y is an integer or double then it returns a new DateTime which is incremented or decremented by specified number of days with a fractional portion corresponding to a portion of the day.

```
Ex: main() {
    Double d = 5.2;
    DateTime dt = new DateTime.now();
    DateTime finalDate = dt -d ;
    System.debug(dt);
    System.debug(finalDate);}
```

Note: Only in case of '+' if X is a string and Y is a string or any other type of non-null argument then it concatenates Y to the end of X.

2. Shorthand Operator:

- a. |= (OR assignment operator):
If X(boolean) and Y(boolean) are both false then X remains false otherwise X is assigned true.
- b. &= (AND assignment operator):
If X(boolean) and Y(boolean) are both true then X remains true otherwise X is assigned false.

3. Equality Operator(==):

If a value of X equals Y, the expression evaluates to true otherwise the expression evaluates to false.

Note: Unlike java(==), in apex it compares object value equality not reference equality except for user-defined types..

Note: String comparison using(==) is case insensitive.

Note: ID comparison using == is case sensitive.

- User-defined types are compared by reference which means that the 2 objects are equal if they reference the same location in the memory.
- You can override this default comparison behavior by equals() and hashCode() methods in your class to compare object values instead.

```
main()
{
    Integer a[] = new Integer[5];
    Integer b[] = new Integer[5];
    for(Integer i=0; i<5; i++)
        a[i]=i+1;
    System.debug(a);
    System.debug(b);
    System.debug(a==b);
}
```

Note: Arrays(==) performs a deep check of all the values before returning its result. Likewise for collection and built-in apex objects.

Note: The comparison of any two values can never result in null though X and Y can be literal null.

Note: SOQL and SOSL use '=' for their equality operator not '=='.

4. Exact Equality Operator(===):

If X and Y reference the exact same location in memory, the expression evaluates to true otherwise false.

```
main()
{
    FirstClass fc = new FirstClass();
    FirstClass fc2 = new FirstClass();
    System.debug(if(fc===fc2));
}
```


5. Exact Inequality Operator(!==):


If X and Y do not reference the exact same location in memory, the expression evaluates to true otherwise false.

6. Relational Operators(<,>,<=,>=):

- a. If X or Y equals null and are integers, doubles, dates or DateTimes then it will return false.
- b. A non-null string or id value is always greater than a null value.
- c. If X and Y are ids then, they must reference the same value otherwise a runtime error occurs.
- d. If X or Y is an id and other value is a string then the string is validated and treated as an id.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

- e. X and Y can not be boolean.

```
Id i1 = "-----";  
Id i2 = "-----";  
System.debug(i1>i2);
```

7. Operator Precedence:

- a. () {} ++ --
- b. -x ! +x
- c. */
- d. +-
e. < > <= >= instanceof
- f. == !=
- g. &&
- h. ||
- i. += == *=



S2 Labs



SALESFORCE HULK

CHAPTER - 3

LOGICS CONTROL AND LOOPING STATEMENT

Logic Control

To control the flow of code execution apex provides conditional statements and loops which helps us execute the code based on conditions or execute the same code repeatedly for a number of times.

In apex, logic control is divided into 3 parts:

1. Conditional Statements
2. Switch Statements
3. Looping Statements

Conditional Statement

IF Statement

Use the if statement to specify a block of code to be executed if a condition is true.

Syntax:

```
if (condition)
{
    // block of code to be executed if the condition is true
}
```

IF Else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax:

```
if (condition)
{
    // block of code to be executed if the condition is true
}
else
{
    // block of code to be executed if the condition is false
}
```

IF Else If Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax:

```
if (condition1)
{
```

```

        // block of code to be executed if condition1 is true
    }
    else if (condition2)
    {
        // block of code to be executed if condition1 is false and condition2 is true
    }
    else
    {
        // block of code to be executed if the condition1 is false and condition2 is false
    }

```

Shorthand IF

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Syntax:

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Looping Statement

Loops can execute a block of code as long as a specified condition is reached.

While Loop

```

while (condition)
{
    // code block to be executed
}

```

Do While Loop

Syntax:

```

do
{
    // code block to be executed
}while (condition);

```

For Loop

Syntax:

```
for(initialization ; condition ; increment/decrement/operation)
{

}
}
```

Example:

```
List<integer> iList = new List<integer>{2,4,6,3,5,8,5,3,5,8,8,5,9};
for(integer a: iList)
```

ForEach loop

Syntax:

```
for (type variable : arrayname)
{

    // code block to be executed

}
```

Example:

```
List<integer> iList = new List<integer>{2,4,6,3,5,8,5,3,5,8,8,5,9};
for(integer a: iList)
{

    a = a*2;

}
```



S2 Labs



SALESFORCE HULK

CHAPTER - 4

SOQL

WHAT IS SOQL?

Salesforce Object Query Language. It is used to retrieve data from Salesforce database according to the specified conditions and objects. Similar to all Apex code, SOQL is also case insensitive.

Use SOQL when you know which objects the data resides in, and you want to:

- Retrieve data from a single object or from multiple objects that are related to one another.
- Count the number of records which meet the specified criteria.
- Sort results as part of the query.
- Retrieve data from number, date, or checkbox fields.

A SOQL query is equivalent to a SELECT SQL statement and searches the org database.

Example:

```
Select name from account // standard object
Select name, Student_name from student__c // custom object
```

But we cannot use asterisk (*) in SOQL queries which we can use in sql.

OR, AND and WHERE Clause:

```
Select Student_name from student__c where couse_opted__c = 'Salesforce Admin';
Select Student_name from student__c where couse_opted__c = 'Salesforce Admin and App Builder'
OR couse_opted__c = 'Salesforce Developer';
Select Student_name from student__c where couse_opted__c = 'Salesforce Admin and App Builder'
OR graduated__c = false;
```


Example:


```
public class soql
{
    public static void main()
    {
        List<Account> accList = [Select Name, NumberOfEmployees from account];
        for(integer i=0; i<accList.size(); i++)
        {
            System.debug(accList[i].numberOfEmployees);
        }


        // *** FOR EACH LOOP ***

        for(Account a:accList)
        {
            System.debug('Acc Name = '+a.Name+' NumOfEmp = ' + a.NumberOfEmployees);
        }
    }
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

```

        '+a.numberOfEmployees);
    }
}

```

Note: It is mandatory to mention the field in the SOQL query which you want to access in the Apex Code.

Note: Comparison of strings are case sensitive using '=' operator in soql.

SOQL Variable Binding:

SOQL queries support Apex variable binding. You can use an Apex variable and filter SOQL records against the value of that variable.

```

String strName = 'John Snow';
List<Position__c> positionList = [SELECT Name
                                FROM Position__c
                                WHERE Name =: strName];

```

SOQL AGGREGATE FUNCTIONS:

“SOQL Aggregate Functions” shows a calculated result from the query and returns **AggregateResult**.

Any query that includes an aggregate function returns its results in an array of **AggregateResult** objects. **AggregateResult** is a read-only sObject and is only used for query results.

1. Sum():

```

AggregateResult ar = [SELECT SUM(Max_Salary__c)
                      FROM Position__c WHERE Max_Salary__c > 10000];

```

2. Max():

```

AggregateResult ar = [SELECT MAX(Max_Salary__c)
                      FROM Position__c
                      WHERE Max_Salary__c > 10000];

```

3. Min():

```

AggregateResult ar = [SELECT MIN(Max_Salary__c)
                      FROM Position__c
                      WHERE Max_Salary__c];

```

4. Count() and Count(fieldName):

```

Integer i = [SELECT count(ID) FROM Account];

```


5. Avg():

```


AggregateResult ar = [SELECT AVG(Min_Salary__c)
                      FROM Position__c];

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 


```
WHERE Max_Salary__c < 5000];
```

6. Count_Distinct():

```
AggregateResult ar = [SELECT COUNT_DISTINCT(Name)
FROM Position__c];
```

SOQL KEYWORDS:

Some of the major keywords used in SOQL queries are:

1. **IN:** IN keyword is used to query on bulk data, which means the WHERE condition involves a collection of records. The collection of records can be a Set, List of IDs, or sObjects which have populated ID fields. Ex:

```
Set<Id> positionIdSet =
    new Set<Id>{'a056000000WZEpg', 'a056000000WZYey'};
List<Position__c> positionList = [SELECT name
FROM Position__c
WHERE Id IN : positionIdSet];
```

Note: Map cannot be used with IN keyword because it is not an iterable collection.

2. **LIKE:** LIKE keyword allows selective queries using wildcards. For example, to query all the position records where Name starts with 'John', you can use a query such as the one given in the code here:

```
List<Position__c> positionList = [SELECT Name
FROM Position__c
WHERE Name LIKE 'John%'];
```

3. **AND/OR:** With AND/OR keywords you can apply AND/OR logic to your query conditions. For example, to query all the position records whose name starts with John and whose Status is open, use query syntax as given here:


```
List<Position__c> positionList = [SELECT name
FROM Position__c
WHERE Name LIKE 'John%' AND Status__c = 'Open'];
```


4. **NOT:** NOT keyword is used for negation. For example, if you want to query all the Position records apart from the two Position Salesforce IDs in a given set, use NOT syntax as given here.

```
Set<Id> positionIdSet =
    new Set<Id>{'a056000000WZEpg', 'a056000000WZYey'};
List<Position__c> positionList = [SELECT name
FROM Position__c
WHERE Id NOT IN : positionIdSet];
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

5. **ORDER BY:** ORDER BY is used to sort the records in ascending(ASC) or descending(DESC) order. It is used after the WHERE clause. Also, you can specify if null records should be ordered at the beginning (NULLS FIRST) or end (NULLS LAST) of the results. By default, null values are sorted first.

```
List<Position__c> positionList = [SELECT Type__c, Comments__c
    FROM Position__c
    WHERE Status__c = 'Open' ORDER BY Type__c ASC NULLS LAST];
```

6. **GROUP BY:** GROUP BY is used with Aggregate functions to group the result set by single or multiple columns. This keyword also groups results for a particular field with minimal code.

```
SELECT Status__c, COUNT(Name)
FROM Position
GROUP BY Status__c;
```

Position Status	Number of Records
Open	24
Close	13
On Hold	7

7. **LIMIT:** LIMIT keyword puts a cap on the number of records a SOQL query should return. This keyword should always be used at the end of the SOQL query.

```
Student__c st1 = [SELECT name FROM student__c
    WHERE student_name__c = 'shrey' limit 1];
```

8. **FOR UPDATE:** FOR UPDATE keyword locks the queried records from being updated by any other Apex Code or Transaction. When the records are locked, other User cannot update them.


```
List<Position__c> positionList = [SELECT Name, Comments__c
    FROM Position__c
    WHERE Status__c = 'Open' FOR UPDATE];
```

9. **ALL ROWS:** ALL ROWS keyword in a SOQL query retrieves all the records from the database, including those that have been deleted. This keyword is mostly used to undelete records from the Recycle Bin.

```
List<Position__c> positionList = [SELECT Name
    FROM Position__c
    WHERE isDeleted = true ALL ROWS];
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 



RELATIONSHIP QUERIES:

Relationship queries involve at least two objects, a parent and a child. These queries are used to fetch data either from Parent object, when SOQL query is written on child, or from child object when SOQL query is written on parent.

1. Child to Parent:

For Standard Objects:

```
SELECT relationship_field.parent_obj_field FROM child_object;
List<Opportunity> oppList = [SELECT name, amount, closeDate, account.name
                             FROM Opportunity];

for(Opportunity o: oppList)
{
    System.debug('Opp: ' + o.name + ' Acc: ' + o.account.name);
}
```

For Custom Objects:

```
SELECT (relationship_field)__r.parent_obj_field FROM child_object;
List<Branch__c> branchList = [select Branch_ID__c, State__c,
                              Branch_of_Bank__r.Bank_Name__c from Branch__c];

for(Branch__c bl: branchList)
{
    System.debug('Bank Name: ' + bl.Branch_of_Bank__r.Bank_Name__c
                + 'Branch: ' + bl.State__c);
}
```

2. Parent to Child Relationship Query:

For Standard Objects:

```
SELECT (SELECT child_obj_field FROM child_relationship_name) FROM
parent_object;
List<Account> accList = [select name, numberOfEmployees, (select name, amount,
                                                           closeDate from opportunities) from Account];

for(Account a: accList)
{
    System.debug('ACC: ' + a.name + ' NoEmpl: ' + a.numberOfEmployees.;
    List<Opportunity> oplist = a.opportunities;
    for(opportunity o: oplist)
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com)

training@cyntexa.com

[Salesforce Hulk](#)

```
{
    System.debug('opportunity name:' + o.name + 'Amount: ' + a.amount);
}
}
```

For Custom Objects:

```
SELECT (SELECT child_obj_field FROM (child_relationship_name)__r)
FROM parent_object;
List<Bank__c> bankList = [select Bank_Name__c, (select Branch_ID__c, Name,
State__c from Branches__r) from Bank__c];
for(Bank__c ba: bankList)
{
    List<Branch__c> branchList = ba.branches__r;
    for(Branch__c br: branchList)
    {
        System.debug('Bank:' + ba.Bank_Name__c + '|' Branch
ID: ' + br.Branch_ID__c
+ '|' Branch Name: ' + br.Name + '|' Branch State: ' + br.State__c);
    }
}
```

Note: You can not write subquery inside a subquery in soql.

3. Multi-level Relationships: In SOQL, you can traverse up to a maximum of five levels when querying data from child object to parent.

```
SELECT Name, Position__r.Contact.Account.Owner.FirstName
FROM Job_Application__c;
SELECT Name, (SELECT Name FROM Job_Applications__r)
FROM Position__c
```

Five Levels:

Job Application → Position → Contact → Account → User

SOQL 'for' Loops:

SOQL 'for' Loops are SOQL statements that can be used inline along with FOR Loops. This means that instead of iterating over a list of records, it is done directly over a SOQL query written inline inside the 'for' Loop iteration brackets.

```
FOR(Position__c pos : [SELECT name, max_salary__c
FROM Position__c
WHERE Status__c = 'Open']) { //code block }
```


There are two formats to this type of SOQL query:


1. Single Variable Format:

Single Variable Format executes the loop once per list of sObject records. The syntax of this format is given here:

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

```
List<Position__c> positionRecordsList = [SELECT name, max_salary__c
                                         FROM Position__c
                                         WHERE Status__c = 'Open'];
FOR(Position__c pos : positionRecordsList) {
    //code block
}
```

2. List Variable Format:

List Variable Format executes the loop code once per 200 sObject records.

```
a.    FOR(Position__c pos : [SELECT name, max_salary__c
                             FROM Position__c
                             WHERE Status__c = 'Open']){
        //code block
    }
b.    FOR(List<Position__c> positionList : [SELECT name, max_salary__c
                                           FROM Position__c]){
        FOR(Position pos: positionList){
            //code block
        }
    }
```

DYNAMIC SOQL:

Dynamic SOQL means creation of SOQL string at runtime with Apex code. It is basically used to create more flexible queries based on user's input.

Use Database.query() to create dynamic SOQL.


```
Public static void main(String str)
{
    String s1 = 'select name from'+str;
    List<sObject> sList = Database.query(s1);
    for(sObject s: sList)
    {
        System.debug(s);
    }
}
```


Note: If the query string is wrong, this method returns the QueryException.

```
Public static void main(String table, String field1)
{
    String s1 = 'SELECT name,';
    S1 = s1+field;
    S1 = s1+'from';
    S1 = s1+table;
    String str = 'where name like \'Acme\'';
    S1 = s1+str;
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](https://www.salesforcehulk.com) 

Note: We can use bind variables (:i) in dynamic SOQL but we can't use bind variable field in a dynamic SOQL. `':500'`

```
Public static void main()
{
    String s = 'Test%';
    Database.query('SELECT Name FROM account where name like :s');
}
```

Note: The above query will work in dynamic as well as static SOQL.

```
Public static void main()
{
    Account a = new Account(name= 'Test', phone='12345');
    Database.query('SELECT Name FROM Account where Phone= :a.phone');
}
```

Note: The above query will not work in dynamic SOQL and will result in a 'variable does not exists' error in it. But it will work perfectly in static SOQL.

There is a hack to use bind variable fields in SOQL.

```
Public static void main()
{
    Account a = new Account(name='abcd', phone='12345');
    String str = a.phone;
    String s = 'SELECT Name FROM Account WHERE phone=:str';
}
```

Note: Use string escape singleQuotes(String str) on the string used for creating the query on dynamic SOQL, just to prevent SOQL injection.


```
String finalString = String.escapeSingleQuotes('SELECT Name FROM Account WHERE
phone=:str');
Database.query(finalString);
```


This method replaces all single quotes (') by (\') which ensures that all single quotation marks are treated as enclosing strings instead of database commands.

```
String fieldString = 'Name';
String sObjectString = 'Position__c';
List<Position__c> positionList = Database.query('SELECT ' + fieldString + ' FROM
' + sObjectString);
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 



S2 Labs



SALESFORCE HULK

CHAPTER - 5

SOSL

WHAT IS SOSL?

Salesforce Object Search Language SOSL is a highly optimized way of searching records in Salesforce across multiple Objects that are specified in the query. A SOSL query returns a list of list of sObjects and it can be performed on multiple objects.

Whether you use SOQL or SOSL depends on whether you know which objects or fields you want to search, plus other considerations.

Use SOSL when you don't know which object or field the data resides in, and you want to:

- Retrieve data for a specific term that you know exists within a field. Because SOSL can tokenize multiple terms within a field and build a search index from this, SOSL searches are faster and can return more relevant results.
- Retrieve multiple objects and fields efficiently where the objects might or might not be related to one another.
- Retrieve data for a particular division in an organization using the divisions feature.
- Retrieve data that's in Chinese, Japanese, Korean, or Thai. Morphological tokenization for CJKT terms helps ensure accurate results.


The basic difference between SOQL and SOSL is that SOSL allows you to query on multiple objects simultaneously whereas in SOQL we can only query a single object at a time.

Some key point:

- In query editor, Find {Test}.
- Find {Test} RETURNING Account(name), Contact(Name).
- Text searches are case-insensitive.
- Find {Test*} where asterisk(*) is a wildcard.
- Find {Test} in Name Fields.
- Whenever referring to ALL FIELDS, use * for emails and when specifying EMAIL fields, no need for *.
- Find {What?} [What type?] [Where?].
- Find {Upsert} RETURNING Account(Name,Phone), Contact(LastName), student__c(Name, Name__c).
- Find {test OR upsert}.
- In order to limit characters
- Find {Test??} ? => single character.
* => zero or multiple character.
- SOSL does not have to use a wildcard (*) as it already matches the fields based on the string match.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

SALESFORCE DEVELOPMENT COURSE

SOSL SYNTAX:

SOSL queries start with the FIND keyword, and then the word or characters that needs to be searched is written in the form of a String. Wildcards can also be used with SOSL query which are not supported by SOQL queries. A wildcard is a keyboard character such as an asterisk (*) or a question mark (?) that is used to represent one or more characters when you are searching for records.

SOSL SUPPORTS TWO WILDCARDS:

```
List<List<sObject>> results = [FIND 'Univ*'....];
List<List<sObject>> results = [FIND 'Jo?n'....];
List<List<sObject>> results = [FIND 'Univ*' IN NAME FIELDS .....];
```

1. **Returning:** If we want to return texts from a particular object then we use returning keyword.

Ex:

```
List<List<sObject>> results = [FIND 'Univ*' IN NAME FIELDS RETURNING Account,
Contact];
List<Account> accounts = (List<Account>)results[0];
List<Contact> contacts = (List<Contact>)results[1];
```

2. **Return specified fields:** If we want to search text in a particular field, we are going to use search group.

Find {contact} IN (searchgroup)

- All Fields (By Default)
- Name Fields
- Email Fields
- Phone Fields

Find {contact} IN (searchgroup) returning objects & fields.

```
List<List<sObject>> results = [FIND 'Univ*' IN NAME FIELDS RETURNING
    Account(Name, BillingCountry), Contact(FirstName, LastName)];
List<Account> accounts = (List<Account>)results[0];
system.debug(accounts[0].Name);
```

Note: Return Type of SOSL is list of list of sObjects which is parent class of all Objects.

Note: In Apex use '' instead of {}.

WHERE Clause:

```
List<List<sObject>> results = [FIND 'Univ*' IN NAME FIELDS
    RETURNING Account(Name, BillingCountry
    WHERE CreatedDate < TODAY())];
```

IN Clause:

Specify which types of text fields to search for on a SOSL query by using the IN SearchGroup optional clause. For example you can search name, email, phone, sidebar, or all fields.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com



training@cyntexa.com



[Salesforce Hulk](#)



Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

SALESFORCE DEVELOPMENT COURSE

You can specify one of the following values (note that numeric fields are not searchable). If not specified the default behaviour is to search all text fields in searchable objects.

Ex:

```
List<List<sObject>> soList = [Find 'Test?' IN ALL FIELDS RETURNING Account, Contact];
List<Account> aList = soList[0];
List<Contact> conList = soList[1];
for(Account a: aList)
{
    System.debug(a);
}
For(Contact c: conList)
{
    System.debug(c);
}
```

Note: In java, class cast exception.

Note: In standard and custom objects, name fields have the NAME FIELD property set to true, in order to make it searchable.

Some standard fields in standard objects already have this property set to true like:

1. Account: Website, Site, NameLocal.
2. Contact: AssistantName, FirstNameLocal, LastNameLocal
3. Lead: Company, FirstNameLocal, LastNameLocal
4. Case: Subject, SuppliedName, SuppliedCompany
5. Event: Subject
6. Task: Subject

Note: If you want to specify a WHERE clause, you must include a field list with at least one specific field.

Find {upsert} Returning Account(Name), Contact, Student__c(Name where Registered_for_course__c='Salesforce' and initial_fees > 60)

ORDER BY Clause:

Find {Test} Returning Account(Name, Phone, WHERE website!=null ORDER BY Name DESC), Contact(Name ORDER BY Name)

LIMIT Clause:

Find {Test} RETURNING Account(Name, Phone LIMIT 50)

Note: Limit should always be the last statement.

Note: Numeric fields are not searchable.

Note: It is recommended to specify the search scope unless you need to search all fields.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com



training@cyntexa.com



[Salesforce Hulk](#)



Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.

SALESFORCE DEVELOPMENT COURSE

SOQL VS. SOSL:

SOQL	SOSL
You know in which object the data you are searching for resides	You are not sure in which object the data might be.
The data needs to be retrieved from single object or related objects.	You want to retrieve multiple objects and fields, which might not be related to each other.
SOQL queries can be used in Classes and Triggers.	They are only supported in Apex Classes and Anonymous blocks.
We can perform DML operations on query result.	We cannot perform DML operations.
It returns records.	It returns fields.
You can count retrieved records.	You cannot count retrieved records.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com



training@cyntexa.com



[Salesforce Hulk](#)



Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs



SALESFORCE HULK

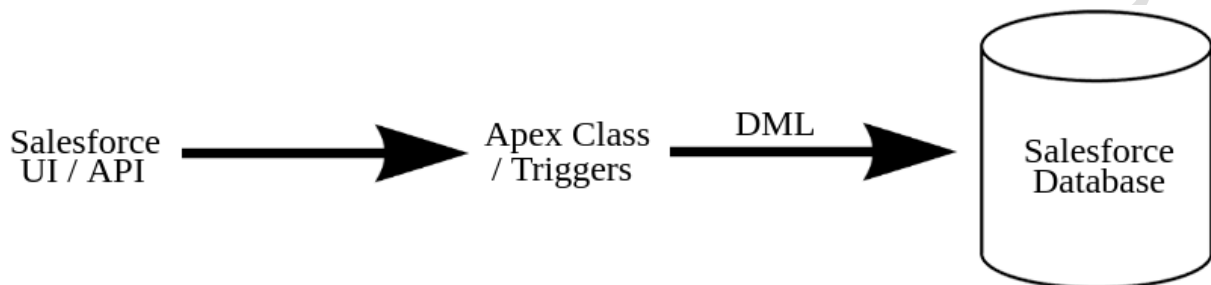
CHAPTER - 6

APEX DML AND DATABASE METHODS

DML STATEMENTS

DML stands for Data Manipulation Language and it enables you to:

- Insert new records in Salesforce
- Update existing records
- Upsert records
- Delete records from the database
- Undelete or restore records from the recycle bin



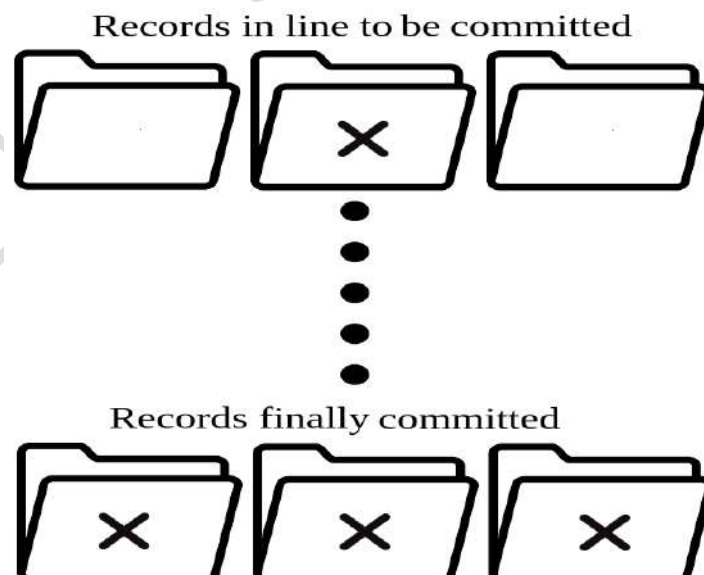
DML Standalone Statements:

- Standalone DML statements can be used to perform DML operations. But if even a single record errors out in the entire list of records then the entire operation is rolled back.

Syntax:


```


insert sObject or list of sObjects
update
delete
upsert
undelete
  
```



By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

1. Insert:

Ex 1:

```
public static void main()
{
    Account a = new Account(Name='Shubham', numberOfEmployees=150);
    insert a;
}
```

Ex 2:

```
List<Contact> newConList = new List<Contact>();
List<Contact> conList = [select firstname, lastname from contact];
for(Contact cc : conList)
{
    Contact c1 = new Contact(lastname=cc.lastname, firstname=cc.firstname);

    insert c1;
} // too many DML statements, hence Limit Reached
for(Contact cc : conList)
{
    Contact c1 = new Contact(lastname=cc.lastname, firstname=cc.firstname);

    newConList.add(c1);
}
insert newConList; // only 1 DML Statement
```

Note: We cannot insert a record in which id is specified.

To create contact and relate to existing account:

```
Account a = [select name from account where name like 'sh%'];
Contact c = new Contact(lastname='Again', Account_id=a.id);
insert c;
```

2. Update:

Ex 1:

```
public static void main()
{
    Account a = [select name from Account where name like 'Sh%' limit 1]
    a.name = 'Cyntexa';
    Update a;
}
```


Ex 2:


```
List<Contact> conList = [select firstname, lastname from contact where
                        createdAt = Today()];

Integer i = 1;
for(Contact cc : conList)
{
    cc.firstname=cc.firstname+i;
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

```
i++;
}
Update conList;
```

Note: We cannot modify a list or collection in a foreach loop where its even elements are iterated. (infinite loop)

3. Upsert:

Ex:

```
List<Contact> newConList = new List<Contact>();
List<Contact> conList = [select firstname, lastname from contact];
Integer i = 1;
for(Contact cc : conList)
{
    String s = cc.name;
    if(s.contains('Contact'))
    {
        cc.phone = '1234';
        newConList.add(cc);
    }
    else
    {
        Contact c1 = new Contact(lastname='Contact'+i);
        i++;
        newConList.add(cc);
    }
}
upsert newconList;
```


4. Delete:


Ex: Delete Duplicate Contacts

```
List<Contact> conList = [select firstname, lastname from contact];
Map<String, ID> mMap = new Map<String, ID>();
for(Contact c: conList)
{
    mMap.put(c.name, c.id);
}
List<Contact> uniqList = new List<Contact>();
List<Contact> delList = new List<Contact>();
Set<String> sSet = mMap.keySet();
Set<ID> uniqSet = new Set<ID>();
for(String s: sSet)
{
    uniqSet.add(mMap.get(s));
}
for(Contact c1: conList)
{
    if((uniqSet.contains(c1.id)))
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

```

        uniqList.add(c1);
    else
        delList.add(c1);
    }
    delete delList;

```

Note: Use isDeleted=true & ALL ROWS keyword to get the deleted records from the recycle bin using SOQL.

Note: ALL ROWS keyword cannot be used in the query editor but can be used in the apex class or function while querying record in [].

5. Undelete:

Ex:

```

List<Contact> deletedContacts = [SELECT name FROM contact
                                WHERE isDeleted=true ALL ROWS];

undelete deletedContacts;

```

6. Merge:

Ex:

```

List<Account> accList = [SELECT name FROM Account LIMIT 3];
Account a = accList[0];
Account b = accList[1];
Account c = accList[2];

List<Account> mergeList = new List<Account>();
mergeList.add(accList[1]);
mergeList.add(accList[2]);

merge a b;           // for 2 records
merge a mergeList;   // for 3 records

```

Note: Merge only works with Accounts, Leads and Contacts.

Note: You can merge 3 records at a time not more than that.

Different Merging Combinations:

- sObject with sObject
- sObject with List<sObject>
- sObject with ID
- sObject with List<ID>

DATABASE CLASS:


```

Public static void main()
{
    List<Contact> conList = new List<Contact>();
}

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

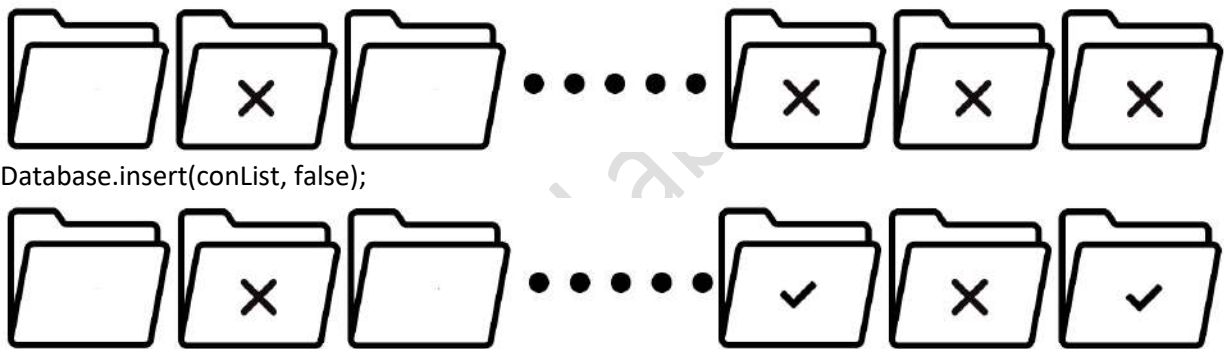

```
for(integer i=0; i<10; i++)
{
    if(i==5)
        conList.add(new Contact());
    else
        conList.add(new Contact(lastname='ABC'+1));
}
insert conList; // Throws DML Exception
Database.insert(conList, false); // Doesn't throw exception if a record fails
}
```

In DML Statements if there is an error or problem in any of the record then the whole DML results in an exception. (DML Exception) which you can handle on your own whereas in database class methods we can specify whether or not to allow partial record passing if errors are encountered.

We can do so by passing an additional boolean parameter.

When this parameter is set as false and if a record fails the remainder of DML operation can still succeed.

Database.insert(conList);



Database Class Methods:

1. Database.insert():

Ex :


```
List<Account> accountsToInsert = new List<Account>();
Account accountToUpdate;
for(Integer i=0; i<3; i++){
    Account acc = new Account(Name = 'Simplilearn', BillingCity = 'New
York');
    accountsToInsert.add(acc);
}
Database.insert(accountsToInsert);
```


2. Database.update():

Ex :

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

```
accountToUpdate = [Select BillingCity FROM Account WHERE Name = 'Simplilearn'
LIMIT 1];
accountToUpdate.BillingCity = 'San Francisco';
Database.update(accountToUpdate);
Account afterUpdate = [Select BillingCity FROM Account WHERE Id =:
accountToUpdate.Id];
System.assertEquals(afterUpdate.BillingCity, 'San Francisco');
```

3. Database.upsert():

Syntax: Database.upsert(sObject/List<sObject>, externalIdField, allOrNone);

Note: If externalIdField left blank then id field will be used as external id.

```
List<Account> accountsList = [SELECT Id, Name, BillingCity
FROM Account WHERE BillingCity = 'San Francisco'];
for (Account acc : accountsList) {
    acc.BillingCity = 'Las Vegas';
}
Account newAccount = new Account(Name = 'SimpliLearn', BillingCity = 'Palo
Alto');
accountsList.add(newAccount);
Database.upsert(accountsList);
```

4. Database.delete():

Ex :

```
List<Account> accList = [SELECT name FROM account limit 10];
Database.delete(accList, false);
```

Note: You cannot delete an account which have cases related to it.

5. Database.undelete():

Ex :

```
List<Account> accList = [SELECT name FROM account
WHERE isDeleted=true ALL ROWS]
Database.undelete(accList, false);
```

6. Database.merge():

Syntax :

Database.merge(master_sObject,
duplicate (sObject/List<sObject/ID/List<ID>), allOrNone);


Ex :


```
List<Account> accList = [SELECT name FROM account LIMIT 3];
Account masterRecord = accList[0];
List<Account> dupList = new List<Account>;
dupList.add(accList[1]);
dupList.add(accList[2]);
Database.merge(masterRecord, dupList);
```

Database class also provides methods which are not available as DML statements such as methods for transaction control & rollback, empty the recycle bin and methods related to SOQL queries.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

1. `emptyRecycleBin(List<ID>/sObject/List<sObject>)`: It permanently deleted the record from recycle bin.

```
List<Account> accList = [SELECT id FROM Account where isDeleted=true ALL ROWS];
Database.emptyRecycleBin(accList);
```

2. `countQuery(String str)`: Returns the number of records that a dynamic soql query would return when executed.

```
Integer i = Database.countQuery('select count() from account');
System.debug(i);
```

Note: `countQuery` will not work if we enter any field inside the count.

`count()` // ok

`count(ID)` // error

Transaction Control and Rollback


3. `Savepoint setSavepoint()`: Returns a savepoint variable that can be stored as a local variable which we can use with a rollback method to restore the database to that point.
4. `Void rollback(Savepoint sp)`: Restores the database to the state specified by the savepoint argument. Any emails submitted since the last savepoint are also rolled back and not sent.


```
Savepoint sp1 = new Database.setSavepoint();
List<Account> accList = [SELECT name, phone FROM account WHERE name like 'Acc%'];
integer i = 1;
for(Account a: accList)
{
    A.phone = i+' ';
    i+=111;
}
update accList;
Savepoint sp2 = Database.setSavepoint();
for(Account b: accList)
{
    b.name=b.name + 'ss';
}
Update accList;
Savepoint sp3 = Database.setSavepoint();
if(roll=1)
    Database.rollback(sp1);
else if(roll=2)
    Database.rollback(sp2);
else if(roll=3)
    Database.rollback(sp3);
```

DATABASE CLASS METHOD RESULT OBJECTS:

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

Database class methods returns the result of data operations. These result object contain useful information about data for each record such as whether the operation is successful or not and any other information.

Each type of operations returns a specific result object type, following are:

Operation	Result Class
Insert, update	Save Result Class
upsert	Upsert Result Class
merge	Merge Result Class
delete	Delete Result Class
undelete	Undelete Result Class
convertLead	Lead Convert Result Class
emptyRecycleBin	Empty Recycle Bin Class Result

DATA BASE CLASS METHODS:

```


Public static void main()
{
    List<Account> accList = new List<Account>();
    for(integer i = 0; i < 10; i++)
    {
        accList.add(new Account(name='Result Test'+i, numberOfEmployees=i));
        accList.add(new Account());
        Insert accList;
    }
    System.debug('Total Accounts:'+accList.size());
    Database.saveResult[] saveList = Database.insert(accList, false);


    for(Database.saveResult s: saveList)
    {
        If (s.isSuccess())
            System.debug('I was successful = '+s.getID());
        Else
        {
            System.debug('I was Unsuccessful = '+s.getID+' because of
            following errors');
            for(Database.error dr: s.getErrors())
            {
                System.debug(dr.getStatusCode()+' '+dr.getMessage());
            }
        }
    }
}

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 



SALESFORCE DEVELOPMENT COURSE




```
        System.debug('fields affected by insertion are :'+  
        +dr.getFields);  
    }  
}  
}
```


Lead Conversion:

```
Public static void main()  
{  
    Lead l = [SELECT name FROM lead WHERE name like 'sh%' LIMIT 1];  
    Lead ll = new Lead(lastName='', companyName='');  
    Insert ll;  
  
    Database.leadConvert lc = new Database.leadConvert;  
    lc.setLeadId(l.id);  
    System.debug('ID of record inserted is: '+l.id);  
    lc.convertedStatus='closed-converted';  
    lc.ownerId='a027F000001XWU7'; // any user id  
  
    Database.leadConvertResult lcr = Database.convertLead(lc);  
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://www.shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs



SALESFORCE HULK

CHAPTER - 7

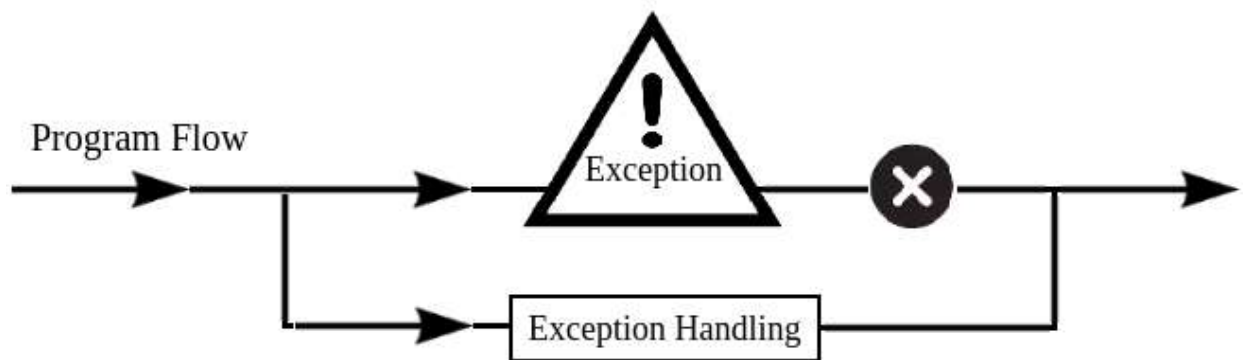
EXCEPTIONAL HANDLING

EXCEPTION HANDLING

An exception denotes an error or event that disrupts the normal flow of code execution.

Exception disrupts the normal flow of the application hence to keep the application running we use exception handling.

Exception handling is the process of responding to exceptions that appear in the code during its runtime so that normal flow of the application can be maintained.



For Example:


DML statements return run-time exceptions if something goes wrong in the database during the execution of DML operations.


You can handle the exceptions in your code by **wrapping your DML statements within try-catch blocks.**

The following example includes the insert DML statement inside a try-catch block:

```
Account a = new Account (Name='Acme') ;
try
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#)



```
{
    insert
}
catch(DmlException e)
{
    // Process exception here
}
```

Exception Statements:

There are different types of exception statements in apex:

1. Throw Statements:

A throw statement is used to generate an exception or to signal that an exception has occurred. The basic syntax of throw statement is shown here:

```
throw exceptionObject;
```

2. Try-catch-finally Statements:

Try, catch, and finally are the Exception statements that can be used together to capture an exception and handle it gracefully.


Try: A try block encloses the code where exception can occur.

Catch: Catch block is optional and comes immediately after the try block and can handle a particular type of exception. A single try statement can have zero or more associated catch statements where each catch statement must have a unique exception type.

Note: Once a particular exception type is caught in one catch block, the remaining catch blocks (if any) are not executed.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



Finally: Finally block is mandatorily executed at the end whether the exception occurred in the try block or not.

It is not compulsory for every try block to have finally associated with it and also there can be at max only 1 finally block that can be associated to 1 try block.

Finally block is generally used to cleanup code or for freeing up resources.

Note: With every try block there should be at least 1 catch or 1 finally block associated.


Let's look at the syntax of try, catch, and finally statements:


```
try
{
    // Try block
}
catch (exceptionType variableName)
{
    // Initial catch block.
    // At least 1 catch block or finally block must be present.
}
catch (Exception e)
{
    // Optional additional catch statement for other exception types.
    // Note that the general exception type, 'Exception' must be the last catch
    block when it is used.
}
finally
{
    // Finally block.
    // this code block is always executed
}
```

Types of Exception:

1. System Defined:

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



- a. **DMLException:** DML Exceptions are exceptions that occur whenever a DML operation fails. This may happen due to many reasons, **the most common one is inserting a record without a required field.**

```
Ex:    try
    {
        Position__c pos = new Position__c();
        insert pos;
    }
    catch(DmlException e)
    {
        System.debug('The following exception has occurred: ' +
            e.getMessage());
    }
```


- b. **ListException:** ListException catches any type of run time error with a list. This list can be of any data type such as integer, string, or sObject.


```
Ex:    try
    {
        List<String> stringList = new List<String>();
        stringList.add('Bhavna');
        // This list contains only one element,
        // but we will attempt to access the second element
        // from this zero-based list.
        String str1 = stringList[0]; //this will execute fine
        String str2 = stringList[1]; // Causes a ListException
    }
    catch(ListException le)
    {
        System.debug('The following exception has occurred: ' +
            le.getMessage());
    }
```

- c. **NullPointerException:** NullPointerException catches exceptions that occur when we try to reference a null variable. Use this exception whenever you are referencing a variable that might turn out to be null.

```
Ex:    try
    {
        String stringVariable;
        Boolean boolVariable = stringVariable.contains('John');
        // Causes a NullPointerException
    }
    catch(NullPointerException npe)
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```
{
    System.debug('The following exception has occurred: ' +
npe.getMessage());
}
```

- d. **QueryException:** QueryException catches any run time errors with SOQL queries. QueryException occurs when there is a problem in SOQL queries such as assigning a query that returns no records or more than one record to a single sObject variable.

Ex:

```
try {
    // This statement doesn't cause an exception,
    // if we don't have a problem in SOQL Queries.
    // The list will just be empty.
    List<Position__c> positionList = [SELECT Name FROM
    Position__c WHERE Name='Salesforce Developer'];
    // positionList.size() is 0
    System.debug(positionList.size());


    // However, this statement causes a QueryException because
    // we're assigning the query result to a Position sObject
    // variable
    // but no Position record is found
    Position__c pos = [SELECT Name FROM Position__c WHERE
    Name='Salesforce Developer' LIMIT 1];
}
catch(QueryException ge) {
    System.debug('The following exception has occurred: ' +
ge.getMessage());
}
```


- e. **Generic Exception:** This exception type can catch any type of exception; that's why it's called generic Exception type. This is used when you are not sure which exception type to use and what exception might occur in the code.

Ex:

```
try {
    List<Position__c> positionList = [SELECT Name FROM
    Position__c WHERE Name='Salesforce Developer'];
    // positionList.size() is 0
    System.debug(positionList.size());
    Position__c pos = [SELECT Name FROM Position__c WHERE
    Name='Salesforce Developer' LIMIT 1];
} catch(Exception ex) {
    System.debug('The following exception has occurred: ' +
ex.getMessage());
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

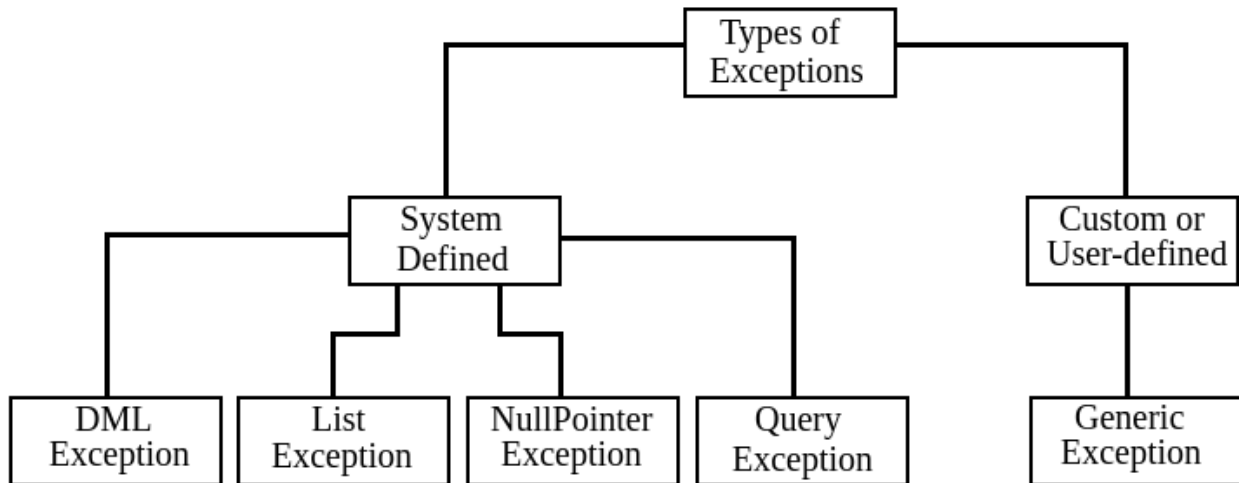
training@cyntexa.com 

[Salesforce Hulk](#)



}

2. Custom or User-defined:



COMMON EXCEPTION METHODS:


All Exception types are Exception classes and these classes have methods that can provide a lot of information about the exception and error that occurred.

1. `getTypeName()`: Returns the type of exception, such as `DmlException`, `ListException`, and `QueryException`.
2. `getMessage()`: Returns the error message and displays for the user.
3. `getCause()`: Returns the cause of the exception as an exception object.
4. `getLineNumber()`: Returns the line number from where the exception was thrown.
5. `getStackTraceString()`: Returns the stack trace as a string.

Ex: try {
 Position__c pos = new Position();
 // Causes an QueryException because
 // we are inserting record without required fields
 insert pos;
 }
 catch(Exception e)
 {

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```
System.debug('Exception type caught: ' + e.getTypeName());
System.debug('Message: ' + e.getMessage());
System.debug('Cause: ' + e.getCause());    // returns null
System.debug('Line number: ' + e.getLineNumber());
}
USER_DEBUG|[6]|DEBUG|Exception type caught: System.QueryException
USER_DEBUG|[7]|DEBUG|Message: List has no rows for assignment to SObject
USER_DEBUG|[8]|DEBUG|Cause: null
USER_DEBUG|[9]|DEBUG|Line number: 2
USER_DEBUG|[10]|DEBUG|Stack trace: AnonymousBlock: line 2, column 1
```

How to Catch Different Exception Types:

You can use several catch blocks—a catch block for each exception type and a final catch block that catches the generic Exception type.

Ex:

```
try {
    Position__c pos = new Position();
    // Causes an QueryException because
    // we are inserting record without required fields
    insert pos;
} catch(DmlException e) {
    System.debug('QueryException caught: ' + e.getMessage());
} catch(QueryException e) {
    System.debug('DMLEException caught: ' + e.getMessage());
} catch(Exception e) {
    System.debug('Exception caught: ' + e.getMessage());
}
```

Custom or User-Defined Exceptions:

- You can create your own top level Exception classes that can have their own member variables, methods, constructors, implement interfaces, and so on.

Ex: `public class MyCustomException extends Exception {}`

- Custom exceptions enable you to specify detailed error messages and have more custom error handling in your catch blocks.

Ex: `public class MyOtherException extends MyCustomException {}`

- You can construct exceptions:

- ◆ With no arguments:


Ex: `new MyException();`


- ◆ With a single String argument that specifies the error message:

Ex: `new MyException('This is bad');`

- ◆ With a single Exception argument that specifies the cause and displays in any stack trace:

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



Ex: new MyException(e);

- ◆ With both a String error message and a chained exception cause that displays in any stack trace:

Ex: new MyException('This is bad', e);

Example:

```
public class PositionException extends Exception {}


public class PositionUtility {
    public static void mainProcessing() {
        try {
            insertPositionMethod();
        } catch(PositionException pe) {
            System.debug('Message: ' + pe.getMessage());
            System.debug('Cause: ' + pe.getCause());
            System.debug('Line number: ' + pe.getLineNumber());
            System.debug('Stack trace: ' + pe.getStackTraceString());
        }
    }
    public static void insertPositionMethod() {
        try {
            // Insert Position record without required fields
            Position__c pos = new Position__c();
            insert pos;
        } catch(DmlException e) {
            // Since Position record is being insert
            // without required fields, DMLException occurs
            throw new PositionException(
                'Position record could not be inserted.', e);
        }
    }
}
```

What exactly happens when an Exception occurs?

- When an exception occurs, code execution halts.
- Any DML operations that were processed before the exception are rolled back and aren't committed to the database.
- Exceptions get logged in debug logs.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



- For unhandled exceptions, that is, exceptions that the code doesn't catch, Salesforce sends an email that includes the exception information.
- The end user sees an error message in the Salesforce user interface.

Exceptions that Can't be Caught

There are some special types of built-in exceptions that can't be caught. Those exceptions are associated with critical situations in the Lightning Platform. These situations require the abortion of code execution and don't allow for execution to resume through exception handling.


Example:

One such exception is the limit exception (**System.LimitException**) that the runtime throws if a governor limit has been exceeded, such as when the maximum number of SOQL queries issued has been exceeded. Other examples are exceptions thrown when assertion statements fail (through System.assert methods) or license exceptions.

Note: When exceptions are uncatchable, catch blocks, as well as finally blocks if any, aren't executed.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#)





S2 Labs



SALESFORCE HULK

CHAPTER - 8

APEX TRIGGERS

INTRODUCTION

Trigger is an Apex Code that executes before or after changes occurs to Salesforce records. These changes includes operations like Insert, Update, Delete, Merge, Upsert and Undelete.

WHEN TO USE TRIGGERS:

1. An Apex Trigger is a Stored Apex procedure that is called whenever a record is inserted, updated, deleted, or undeleted. **If any change happens to a single or multiple records**, the Apex Trigger created on that object will be fired.
2. For example, we can have a trigger run:
 - ◆ Before an object's record is inserted into the database.
 - ◆ After a record has been deleted.
 - ◆ Even after a record is restored back from recycle bin.
3. **Use triggers to perform tasks that can't be done using a point & click tool in Salesforce. If the task is possible using point & click tools then always prefer doing it from them.**
4. Triggers are active by-default when created and Salesforce automatically fires active triggers when specified database event occurs.


TYPES OF APEX TRIGGERS:


1. Before Triggers: These are used to update/modify or validate records before they are saved to database.
2. After Triggers: These are used to access fields values that are set by the system like **recordId**, **lastModifiedDate field**. Also, we can make changes to other records in the after trigger but not on the record which initiated/triggered the execution of after trigger because the records that fire the **after trigger are read-only**.

Syntax & Trigger Events:

```
trigger TriggerName on ObjectName (trigger_events)
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```
{  
    //code-block  
}
```

where **trigger-events** can be a comma-separated list of one or more of the following events:

- Before insert
- Before update
- Before delete
- After insert
- After update
- After delete
- After undelete

Example:

```
Trigger firstTrigger on Account(before insert)  
{  
    System.debug('I am before insert.');
```

```
}
```

```
Trigger secondTrigger on Account(after insert)  
{  
    System.debug('I am after insert.');
```

```
}
```

For example:


```
Update Account a;  
(Before update Trigger of Account a)  
{  
    Inserts contact b;  
}  
(after insert of contact B)  
{  
    Insert a; // over here we are updating Account a in its before trigger  
            // indirectly i.e. not allowed.  
}
```

Note: If you update or delete a record in its before trigger, or delete a record in its after trigger you'll receive an error and this includes both direct and indirect operations.

```
trigger ApexTrigger on Account (before update)  
{  
    //See trigger3
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```
Contact c = new Contact(LastName = 'Steve');  
insert c;  
}
```

```
trigger ApexTrigger on Contact (after insert)  
{  
    Account a = [Select Name from Account Limit 1];  
    a.name = 'Updated Account';  
    MyException me = new MyException();  
    throw me;  
    //update a; // Not Allowed  
}
```


Triggers can modify other records of the same type as the records that initially fired the trigger. As triggers can cause other records to change and because these changes can, in turn, fire more triggers, the apex runtime engine considers all such operations a single unit of work & sets limits on the number of operations that can be performed to prevent infinite recursion.


TRIGGER ORDER OF EXECUTION:

In Salesforce there is a particular order in which events execute whenever a record is changed or inserted. Following are the steps of trigger execution:

- Step 1:
Load the original record or initialize on insert.
- Step 2:
Override the old record values with the new values.
- Step 3:
Execute all before triggers.
- Step 4:
Run the system & user defined validation rules.
- Step 5:

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#)



Save the record **but do not commit the record to the database.**

Step 6:

Execute all after triggers.

Step 7:

Execute the assignment rules.

Step 8:

Execute the auto-response rules.

Step 9:

Execute the workflow rules.

Step 10:

If there are workflow field updates then execute the field update.

Step 11:

If the record was updated with a workflow field update then execute before and after triggers created on the object in the context again **but only once.**

Step 12:

Execute the processes on that record.

Step 13:

Execute the escalation rules.

Step 14:

Update the roll up summary fields & cross object formula fields.

Step 15:

Repeat the same process with the affected parent or grand-parent records.

Step 16:

Evaluate criteria based sharing rules.

Step 17:

Commit all DML operations to the database.

Step 18:

Execute post commit logic such as sending emails.


TRIGGER CONTEXT VARIABLE:

Context Variables allow developers to access run-time context of any trigger.

These context variables are contained in System.Trigger class.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



Types of context variable:

1. **Trigger.new:**

```
trigger ApexTrigger on Account(before insert)
{
    for(Account acc: Trigger.new)
    {
        acc.NumberOfEmployees = 100;
    }
}
```

Note: This sObject list is only available in **insert, update & undelete triggers** and records can only be modified in before triggers.


2. **Trigger.old:** This variable returns a list of the old version of sObject records. This list is only available in **update and delete triggers**.


```
trigger ApexTrigger on Opportunity (before update)
{
    // Only available in Update and Delete Triggers


    for(Opportunity oldOpp: Trigger.old)
    {
        for(Opportunity newOpp: Trigger.new)
        {
            if(oldOpp.id == newOpp.id && oldOpp.Amount != newOpp.Amount)
                newOpp.Amount.addError('Amount cannot be changed'); // Trigger Exception
        }
    }
}
```

3. **Trigger.newMap:** This variable returns a map of ids to the new versions of sObject records.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 

Note: This map is only available in **before update, after insert, after update and after undelete triggers**.

trigger ApexTrigger on Account (before update) {

// Only available in beforeUpdate, afterUpdate, afterInsert, afterUndelete Triggers

Map<Id,Account> nMap = new Map<Id,Account>();
nMap = trigger.newMap;

List<Contact> cList = [Select LastName from Contact where AccountId in :nMap.keySet()];

```
for(Contact c: cList)
{
    Account a = nMap.get(c.AccountId);
    c.MailingCity = a.BillingState;
}

update cList;
}
```


4. **Trigger.oldMap**: A map of IDs to the old version of the sObject records.


trigger ApexTrigger on Opportunity (before update) {

// Only available in Update and Delete Triggers

```
Map<Id,Opportunity> oMap = new Map<Id,Opportunity>();
oMap = trigger.oldMap;
for(Opportunity newOpp : trigger.new)
{
    Opportunity oldOpp = new Opportunity();
    oldOpp = oMap.get(newOpp.Id);
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```
if(newOpp.Amount != oldOpp.Amount)
{
    newOpp.Amount.addError('Amount cannot be changed'); // Trigger Exception
}
}
```

Note: This map is only available in **update & delete** triggers.


5. isInsert: Returns true if the trigger was fired due to an insert operation.
6. isUpdate: Returns true if the trigger was fired due to an update operation.
7. isDelete: Returns true if the trigger was fired due to a delete operation.
8. isUndelete: Returns true if the trigger is fired after a record is recovered from recycle bin.
9. isBefore: Returns true if the trigger was fired before any record was saved.
10. isAfter: Returns true if the trigger was fired after all records were saved.
11. Size: Returns the total number of records in a trigger invocation, both old and new.


trigger ApexTrigger on Contact (before insert, before update, before delete, after insert, after update, after delete) {


```
// 5) isInsert
// 6) isUpdate
// 7) isDelete
// 8) isUndelete
// 9) isBefore
//10) isAfter
//11) Size

if(trigger.isBefore)
{
    if(trigger.isInsert)
    {
        System.debug('I am before Insert');
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 



```
} else if(trigger.isUpdate)
{
    System.debug('I am before Update');
} else if(trigger.isDelete)
{
    System.debug('I am before Delete');
}
} else if(trigger.isAfter)
{
    if(trigger.isInsert)
    {
        System.debug('I am after Insert');
    } else if(trigger.isUpdate)
    {
        System.debug('I am after Update');
    } else if(trigger.isDelete)
    {
        System.debug('I am after Delete');
    } else if(trigger.isUndelete)
    {
        System.debug('I am after Undelete');
    }
}
}
```

Points to consider with Trigger Context Variables:

- Trigger.new and Trigger.old cannot be used in Apex DML Operations.
- You can use an sObject to change its own field values using trigger.new but only in before triggers.
- Trigger.old is always read-only.
- You can not delete Trigger.new.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com)

training@cyntexa.com

[Salesforce Hulk](#)

- Upsert and merge events do not have their own triggers, instead they fire other triggers based on the events as a result of merge operation.

EXECUTION ORDER OF UPSERT & MERGE OPERATION:

- In case of upsert operation:
- ◆ The records which are getting inserted fires before & after insert triggers.
 - ◆ The records which are getting updated fires before and after update triggers.
- In case of merge operation:
- ◆ The records which are deleted due to merge operation fires a single delete event (that means before & after delete will be fired only once, irrespective of the number of records deleted).
 - ◆ The master record fires a single update event (before & after update) but no triggers are fired for the child records which are reparented as a result of merge operation.

Order of Events in a merge operation:

1. Before delete trigger fires.
2. Deletes record from database.
3. After delete trigger fires.
4. Before update for master record fires.
5. Record gets updated.
6. After update for the master record fires.

TRIGGER EXCEPTIONS:


Triggers can be used to prevent DML operations from occurring by calling the `addError()` method on a record or field. When used on `trigger.new` records in insert & update triggers and on `Trigger.old` records in delete triggers, the custom error message is displayed in the application interface & logged to.

Example:

```
Trigger AccountDeletion on Account (before delete) {
    for (Account a: [SELECT id FROM Account where id in (SELECT AccountId from
        opportunities) AND id in :Trigger.old]) {
        Trigger.oldMap.get(a.id).addError('Cannot delete account with related
        opportunities.');
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



Note: If a trigger even throws an unhandled exception, all records are marked with an error & no further processing takes place.

BEST PRACTICE WITH TRIGGERS:

Bulkified Triggers:

Bulkifying triggers means writing apex triggers using bulk design pattern so that triggers have better performance and consume less server resources.

As a result of it bulkified code can process large number of records efficiently and run within governor limits on force.com platform.

The main characteristics of bulkified code is:

1. Operating on all records of trigger.
2. Performing SOQL & DML on collections of sObjects instead of single sObject at a time.
3. Using maps to hold query results organized by record id. Avoid query within a query and save records in map which later can be accessed through map rather than using SOQL.
4. Using Sets to isolate distinct records.

trigger ApexTrigger on Lead (before insert)

```
{  
    /**** Bulkified Triggers ***/
```


```
    Lead l = Trigger.new[0]; // will only update the value for first record  
    l.Rating = 'Warm'; //Avoid using Triggers like this
```

```
    //Always use Triggers in Bulkified form
```

```
    for(Lead l : Trigger.new) // will iterate through all the new records
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```

{
    l.Rating = 'Warm';
}

}

```

Bulkified Triggers - Coding practices:

Example 1:

Avoid creating Triggers that work only for individual records but not for entire datasets:

```

trigger testTrigger on Account__c (before insert) {
    Account__c acc = Trigger.New[0];
    acc.Address__c = 'Temporary Address';
}

```

Create Triggers that use loops to help iterate over a list of records within a transaction:

```

trigger testTrigger on Account__c (before insert) {
    integer i = 1;
    for(Account__c acc : Trigger.new){
        acc.Address__c = 'Test Address '+i;
        i++;
    }
}

```

Example 2:

In this code, let's assume 200 Account records are updated, so the "for" loop would iterate over 200 records.

```

trigger BranchTrigger on Branch__c (before update) {
    for(Branch__c br : Trigger.new){
        List<Account__c> accList = [SELECT Name, Account_Name__c, Address__c,
            Balance__c FROM Account__c
            WHERE Account_of_Branch__c = :br.id];
        System.debug(accList);
        // Perform specified logic with queried records
    }
}

```

Now let's look at a good example of querying bulk data and iterating it.


```

trigger BranchTrigger on Branch__c (before update) {
    List<Account__c> accList = [SELECT Name, Account_Name__c, Address__c,

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



```

        Balance__c FROM Account__c
        WHERE Account_of_Branch__c IN :Trigger.New];
    System.debug(accList);
    for(Account__c acc : accList){
        // Perform specified logic with queried records
    }
}

```

Example 3:

DML statements are also bound by Governor Limits; you can call only 150 DML operations in a transaction.

```

trigger BranchTrigger on Branch__c (before update) {
    List<Account__c> accList = [SELECT Name, Account_Name__c, Address__c,
        Balance__c FROM Account__c
        WHERE Account_of_Branch__c IN :Trigger.New];

    System.debug(accList);
    integer i = 0;
    for(Account__c acc : accList){
        acc.Address__c = 'Test Address '+i;
        i++;
        update acc;
    }
}

```

Let's look a correct coding example where we have instantiated another Account object list called "accToUpdate."

```


trigger BranchTrigger on Branch__c (before update) {
    List<Account__c> accToUpdate = new List<Account__c>();
    List<Account__c> accList = [SELECT Name, Account_Name__c, Address__c,
        Balance__c FROM Account__c
        WHERE Account_of_Branch__c IN :Trigger.New];

    System.debug(accList);
    integer i = 0;
    for(Account__c acc : accList){
        acc.Address__c = 'Test Address '+i;
        i++;
        accToUpdate.add(acc);
    }
    if(!accToUpdate.isEmpty()){
        update accToUpdate;
    }
}

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



Trigger Helper Class Pattern:

According to “Best Practices” suggested by Salesforce, we should always use a Helper Class (Apex Class) with a Trigger.

It is a design pattern which makes it easy to maintain the code in the long term.

Common Avoidable Practice:

Salesforce record changes → Trigger containing all the performing code, executes → End

Best Practice:

Salesforce record changes → Trigger calls out to one or multiple classes → Class contains the performing code which executes → End


Example:

Let's understand the concept of Trigger Helper Class Pattern with an example of a Trigger on Account object, which fires for all the Trigger events.

```
trigger accUpdate on Account (before insert, after insert, before update, after update)
{
    if (Trigger.isBefore)
    {
        If (Trigger.isInsert) {
            // execute first trigger
            AccTriggerHelper.firstMethod(Trigger.new);
            // execute second trigger
            AccTriggerHelper.secondMethod(Trigger.new);
            // both of these trigger will follow the execution order
        }
        Else if (Trigger.isUpdate)
        {
            // write the code for before update
        }
        Else if (Trigger.isDelete)
        {
            // write the code for before delete
        }
        Else if (Trigger.isUndelete)
        {
        }
    }
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#)




```
        // write the code for before undelete
    }
}
Else if (Trigger.isAfter)
{
    If (Trigger.isInsert)
    {
        // write the code for after insert
    }
    Else if (Trigger.isUpdate)
    {
        // write the code for after update
    }
    Else if (Trigger.isDelete)
    {
        // write the code for after delete
    }
    Else if (Trigger.isUndelete)
    {
        // write the code for after undelete
    }
}
}
```


Handling Recursion in triggers:

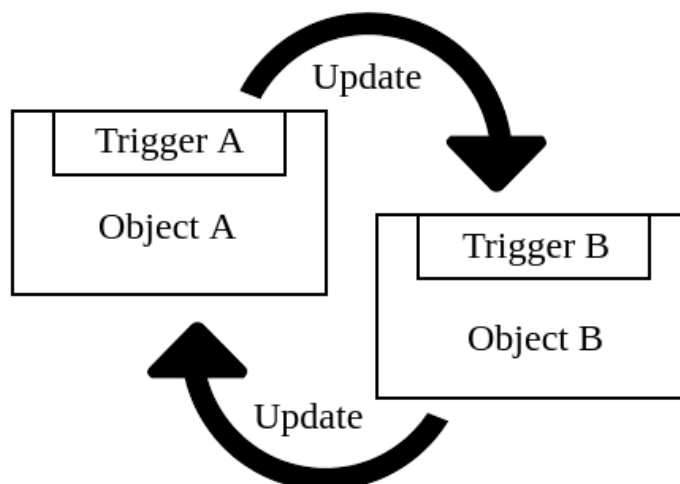
Recursion in Triggers happens when a Trigger is called repeatedly, resulting in an infinite loop.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#) 



Single Transaction

To counter recursion, we need to:

1. Create another class called RecursiveTriggerHandler.
2. Make use of "Static" variables.

Example:

```
public class RecursiveTriggerHandler {  
    public static Boolean isFirstRun = true;  
}  
trigger BranchTrigger on Branch__c (before update) {  
    if(RecursiveTriggerHandler.isFirstRun){  
        RecursiveTriggerHandler.isFirstRun = false;  
        // Call Helper Class method  
        BranchTriggerHelper.firstMethod(trigger.new);  
    }  
}
```


The first instance of the Trigger is run and if this variable is true, the logic in the Helper Class executes.

OTHER BEST PRACTICES FOR WRITING TRIGGERS:

1. Always create only one Trigger per object.
2. Create logic-less Triggers and use Helper Class Design Pattern in which the helper class will contain all the logic.
3. Create context specific handler methods in the Helper Class.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com) 

training@cyntexa.com 

[Salesforce Hulk](#)



4. Bifurcate “insert” and “update” Trigger logic contexts and create two different methods in the Trigger’s helper class.

Example:

```
trigger PositonTrigger on Position__c (after insert, after update) {
    if(Trigger.isAfter && Trigger.isInsert) {
        PositionTriggerHandler.handleAfterInsert(Trigger.new);


    } else if(Trigger.isAfter && Trigger.isUpdate) {
        PositionTriggerHandler.handleAfterUpdate(Trigger.new, Trigger.old);
    }
}
```

Some important points about triggers:

- The order of execution isn’t guaranteed when having multiple trigger for same object with same events.
- When a DML call is made with “Partial Success allowed” then more than one attempts can be made to save the successful records if the initial attempts results in errors for the same records.
- Some operations do not invoke triggers:
 - Cascade Delete
Records that don’t initiate a delete event don’t cost trigger execution.
 - Cascade update of child records that are re-parented as a result of merge operation.
 - Mass campaign status changes.
 - Update account triggers do not fire before and after a business account record type is changed to a person account record type or vice versa.
 - Before trigger associated with the following operations are fired during lead conversion only if validations and triggers for lead conversion are enabled in the organization
 - Insert an account, contact & opportunity
 - Update of account and contacts.
- Fields not available in before insert triggers
 - Opportunity.Amount
 - Opportunity.isWon

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#)





- Opportunity.isClosed
- ID (for all records)
- CreatedDate and LastModifiedDate
- Fields not editable in after triggers
 - Event.WhoID
 - Task.WhoID
- When opportunity has no line items then amount can be modified by a before trigger.

By: Shrey Sharma | Mobile: +91 75 6869 7474

[ShreySharma.com](https://shreysharma.com)

training@cyntexa.com

[Salesforce Hulk](#)



S2 Labs



SALESFORCE HULK

CHAPTER - 9

APEX TESTING



SALESFORCE DEVELOPMENT COURSE



WHAT IS APEX TESTING?

Apex testing is the process that helps you to create relevant test data in the test classes and run them in Salesforce. As a part of Apex testing, you need to test for:

1. Positive behavior: Test Apex code to verify that it works as per specification when appropriate input is provided to the code.
2. Negative behavior: Test the limitations of the system if a negative value, unexpected value, or input is provided to the Apex code
3. Restricted user: Check that a user whose access to certain objects or records is restricted, is barred from accessing them.

UNDERSTANDING TESTING IN APEX:

1. Apex provides a testing framework that allows you to write unit tests, run your tests, check test results, and have code coverage results.
2. Testing is the key to successful long term development and is a critical component of the development process. It is strongly recommended that you use a test-driven development process, that is, test development which occurs at the same time as code development.
3. Testing is key to the success of your Application, particularly if your application is to be deployed to customers. If you validate that your application works as expected, that there are no unexpected behaviours, your customers are going to trust you more.
4. There are two ways of testing an application:
 - ◆ One is through the Salesforce user interface, important, but merely testing through the UI will not catch all the use cases for your application. [MANUAL]
 - ◆ The other way is to test for bulk functionality through your code if it's invoked using SOAP API or by Visualforce standard set controller. [Automated]
5. Before you can deploy your code or package it for the Force.com AppExchange, the following must be true:
 - ◆ **At least 75% of your Apex Code must be covered by unit tests, and all of those tests must complete successfully.**
 - ◆ Every trigger must have some test coverage.
 - ◆ All classes and triggers must compile successfully.
6. Salesforce runs all tests in all organisations that have Apex code to verify that no behavior has been altered as a result of any service upgrade.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



7. Note the following:

- ◆ When deploying Apex to a production organization, each unit in your organization's namespace is executed by default.
- ◆ Calls to System.debug are not counted as a part of code coverage.
- ◆ Test methods and test classes are not counted as a part of Apex Code coverage.

Purpose of Apex Testing:

- To ensure that all the configuration you have built in Salesforce and your Application works as per the specification.
- To create a robust, long lasting system that your customers can rely on.
- To ensure there are no scenarios that are unhandled.
- To ensure that there are no errors which you might have missed in the code.
- To find any errors or unexpected behaviors in the code.

NOTE: In Salesforce, if the Apex code is not properly tested, it can't be deployed into the Production environment.

What are Apex Unit Tests & How to write them?

Unit tests are class methods that verify whether a particular piece of code is working properly or not.

Unit test methods take no arguments, commit no data to the database, send no emails, and are flagged with the testMethod keyword or the **isTest** annotation in the method definition.

Also, test methods must be defined in test classes , that is, classes annotated with isTest.

Create different different test methods to test different functionalities.

In each test method write different test cases to test your code whether it is working properly with the different inputs or not.


After API version 32 test method is used as @istest.


For ex:

```
@isTest
Private class myClass {
    Static void testMethod myTest {
        // code block
    }
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



```

    }
}
@isTest
Private class myClass {
    @isTest
    static void myTest() {
        // code block
    }
}

```

Use the isTest notation to define classes and methods that only contain code used for testing your application. The isTest annotation on methods is equivalent to the testMethod keyword.

Note: Classes defined with the isTest annotation don't count against your organization limit of 3 MB for all Apex code.

This is an example of a test class that contains two test methods:

```

@isTest
Private class MyTestClass {
    // Methods for testing
    @isTest static void test1() {
        // Implement test code
    }
    @isTest static void test2() {
        // Implement test code
    }
}

```

- **Classes and methods defined as isTest can be either private or public.** The access level of the test methods **doesn't matter**. This means you don't need to add an access modifier when defining a test class or test methods. **The default access level in Apex is private.** The testing framework can always find the test methods and execute them, regardless of their access level.
- **Classes defined as isTest must be top-level classes and can't be interfaces or enums.**
- Methods of a test class can only be called from a running test, that is, a test method or code invoked by a test method, and can't be called by a non-test request.

Characteristics of Unit test Methods:

Some characteristics of unit test methods are that they:

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



- Are contained in separate classes which are called as Test Classes specifically created to run tests.
- **Do not commit any data to the database.**
- Do not send any emails.
- Are always flagged with the keyword `testMethod` or `@isTest` annotation at the method definition level.
- Must always be defined in a test class. This test class should be annotated with `@isTest` annotation.
- **Are always defined as static methods.**

SYSTEM-DEFINED METHODS USED IN TEST CLASSES

The common system-defined unit test methods are:

1. `startTest`: `startTest` method marks the point in your test code when the test actually begins.
2. `stopTest`: `stopTest` method comes after `startTest` method and marks the end point of an actual test code.

Purpose:

Any code that executes after the call to `start test()` and before `stop test ()` is assigned a new set of governor limits.


Any code that executes after the call to `stop test()` is arranged the original limits that were in effect before `start test()` was called.


```
Ex:
@isTest
private class myClass {
    static testMethod void myTest() {

        // Create test data
        .....
        .....
        Test.startTest();
        // Actual apex code testing
        .....
        Test.stopTest();
    }
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



3. **isRunningTest:** This method returns true if the currently executing code was called by code contained in a test method, otherwise false.

Ex:

```
public class PositionTriggerHandler{

    public static void processUpdatedRecords(List<Position__c> updateList) {
        if (Test.isRunningTest()) {
            //unit testing alternative code goes here
        }
        else {
            //execute normally
        }
    }
}
```

4. **setFixedSearchResults:** This method defines a list of fixed search result IDs which are then returned as results in SOSL statements in a test method.

Ex:

```
@isTest
private class SoslFixedResultsTest1 {

    public static testMethod void testSoslFixedResults() {
        Id [] fixedSearchResults= new Id[1];
        fixedSearchResults[0] = 'a00x0000003G89h';
        Test.setFixedSearchResults(fixedSearchResults);
        List<List<SObject>> searchList = [FIND 'Salesforce'
            IN ALL FIELDS RETURNING
            Position__c(id, name WHERE name = 'Salesforce' LIMIT 1)];
    }
}
```

5. **setCurrentPage:** `setCurrentPage` and `setCurrentPageReference` are Visualforce test methods that set the current PageReference for the Visualforce controllers.
6. **setCurrentPageReference:** We will learn about these methods in the coming chapters.

Unit Test Considerations:

Here are some things to note about unit tests.

- Test methods can't be used to test Web service callouts. Instead, use mock callouts such as `Test Web service callouts` and `Testing Http Callouts`.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs

SALESFORCE DEVELOPMENT COURSE



- You can't send email messages from a test methods.
- Since test methods don't commit data created in a test, you don't have to delete test data upon completion.

Accessing Private Test Class Members:

If class Members are private, they aren't visible to the test class.

- I. Call the private method in the public method which is already exposed to the test class.
- II. Annotated with these class member with **@testvisible** which allows them to be accessed by test methods and only code running in test context.

For Example:

```
Public class VisibleSampleClass {
    // Private member variables
    @TestVisible private integer recordNumber = 0;
    @TestVisible private string areaCode = '(415)';
    // Public member Variable
    Public integer maxRecords = 1000;

    // Private inner class
    @TestVisible class Employee {
        String fullName;
        String phone;

        // Constructor
        fullName = s;
        Phone = ph;
        @TestVisible Employee(String s, String ph) {
        }
    }
    // Private method
    @TestVisible private string privateMethod(Employee e) {
        System.debug('I am private');
        recordNumber++;
        String phone = areCode + ' ' + e.phone;
        String s = e.fullName + '\s phone number is'+ phone;
        System.debug(s);
        return s;
    }
    // Public method
    Public void public method() {
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



```

        maxRecords++;
        System.debug('I am public');
    }

    // Private custom exception class
    @TestVisible private class MyException extends Exception {}
}

```

Note: Test methods aren't allowed in non-test classes, you must move the test methods from the old class into a new test class (a class annotated with `isTest`) when you upgrade the api version of your class. You might run into visibility issues when accessing private methods or member variables of the original class.

In this case, just annotate these private members with `TestVisible`.

WHAT IS TEST DATA?

- Test data is the transient data that is not committed to the database and is created by each test class to test the Apex code functionality for which it is created. Use of this transient test data makes it easy to test the functionality of other Apex classes and triggers.
- This does not change the data already in the system, nor is any cleanup required after testing has been finished.
- **Test classes can be moved to any environment and executed there as they create their own data; hence, they are environment or org agnostic.**

Ex:

```

trigger PositionTrigger on Position__c (before update) {
    if (Trigger.isBefore && Trigger.isUpdate) {
        PositionTriggerHandler.handleBeforeUpdate(Trigger.newMap,
        Trigger.oldMap);
    }
}


public class PositionTriggerHandler {
    //static method called to handle the before update logic
    public static void handleBeforeUpdate(Map<Id, Position__c> newMap, Map<Id,
    Position__c> oldMap) {
        //Loop over all the position records being updated
        for (Position__c pos : newMap.Values()) {
            //Compare new Min Salary field value with old value
            if (pos.Min_Salary__c <> oldMap.get(pos.Id).Min_Salary__c) {
                //If old and new values are not same, assign old value to comments field
            }
        }
    }
}

```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



```

        pos.Comments__c = String.valueOf(oldMap.get(pos.Id).Min_Salary__c);
    }
}
}
}
}
}
@isTest
public class PositionTriggerTest {
    static testMethod void updateMinimumSalary(){
        PositionTriggerTest.insertPositionRecord();
        List<Position__c> positionListToUpdate = new List<Position__c>();
        for(Position__c pos : [Select Min_Salary__c from Position__c]){
            pos.Min_Salary__c = 30000;
            positionListToUpdate.add(pos);
        }
        Test.startTest();
        update positionListToUpdate;
        Test.stopTest();
        List<Position__c> posList = [Select Comments__c from Position__c];
        system.assertEquals(posList[0].Comments__c, '20000');
    }
    static void insertPositionRecord(){
        List<Position__c> positionListToInsert = new List<Position__c>();
        for(Integer i=0; i< 100; i++){
            Position__c pos = new Position__c();
            pos.Name = 'Salesforce Developer'+i;
            pos.Min_Salary__c = 20000;
            positionListToInsert.add(pos);
        }
        insert positionListToInsert;
    }
}

```

Features of isTest(SeeAllData=true) Annotation:

By default, Salesforce org's data is not visible to the test class, rather it has to create its own data. However, this behavior can be changed by annotating a Test class or test method with @isTest(SeeAllData=true) annotation opens up the entire org's data to the test class.

Characteristics of the IsTest(SeeAllData=true) Annotation are:

- If this annotation is defined on the Class level, then this annotation is automatically applied to all its test methods. This annotation can also be used individually at the method level.
- If the annotation is applied on the class level, then using isTest(SeeAllData=false) on a particular method does not restrict its access to the org's data.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



Features of isTest(SeeAllData=true) Annotation:

- Use Test.getStandardPricebookId() method to fetch standard pricebook id in Test Class.
- No need to write a SOQL query.
- No need to use SeeAllData annotation.

Ex:

```
@isTest
public class PriceBookTest {
    // Utility method that can be called by Apex tests to create price book entries.
    static testmethod void addPricebookEntries() {
        // First, set up test price book entries.
        // Insert a test product.
        Product2 prod = new Product2(Name = 'Laptop X200',
            Family = 'Hardware');
        insert prod;
        // Get standard price book ID.
        // This is available irrespective of the state of SeeAllData.
        Id pricebookId = Test.getStandardPricebookId();
        // 1. Insert a price book entry for the standard price book.
        // Standard price book entries require the standard price book ID we got earlier.
        PricebookEntry standardPrice = new PricebookEntry(
            Pricebook2Id = pricebookId, Product2Id = prod.Id,
            UnitPrice = 10000, IsActive = true);
        insert standardPrice;
    }
}
```

Steps to Load Test Data:

To load data in a test class without writing many codes, follow these three steps:

- Create data you want to load in a .csv file.
- Create a static resource and upload the .csv file in it.
- Call Test.loadData in your test method.

Syntax:

```
List<sObject> ls = Test.loadData(Position__c.sObjectType, 'positionResource');
```

Loading Test Data—Example:

Here is an example for Position object data load via Test.loadData method. Static Resource name is “positionResource” and Test Class is “PositionTriggerTest”.

Ex:

```
@isTest
public class PositionTriggerTest {
```



SALESFORCE DEVELOPMENT COURSE



```
static testMethod void updateMinimumSalary(){
    // Load the test accounts from the static resource
    List<SObject> positionLoadRecordList = Test.loadData(Position__c.sObjectType,
'positionResource');
    // Verify that all 9 test position records were created
    system.assertEquals(positionLoadRecordList.size(), 9);
    List<Position__c> positionListToUpdate = new List<Position__c>();
    for(Position__c pos : [Select Min_Salary__c from Position__c]){
        pos.Min_Salary__c = 30000;
        positionListToUpdate.add(pos);
    }
    Test.startTest();
        update positionListToUpdate;
    Test.stopTest();
}
}
```

Test Setup Methods:

These are Special methods which are used to create test data that becomes accessible to the whole Test Class.

These methods use @TestSetup annotation.

Ex:

```
@isTest
private class CommonTestSetup {
    @testSetup static void setup() {
        // Create common test accounts
        List<Account> testAccts = new List<Account>();
        for(Integer i=0;i<2;i++) {
            testAccts.add(new Account(Name = 'TestAcct'+i));
        }
        insert testAccts;
    }
    @isTest static void testMethod1() {
        // Get the first test account by using a SOQL query
        Account acct = [SELECT Id FROM Account WHERE Name='TestAcct0' LIMIT 1];
        // Modify first account
        acct.Phone = '555-1212';
        // This update is local to this test method only.
        update acct;
        // Delete second account
        Account acct2 = [SELECT Id FROM Account WHERE Name='TestAcct1' LIMIT 1];
        // This deletion is local to this test method only.
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



```
delete acct2;  
}
```


UNIT TEST BEST PRACTICES:

- Cover as many lines of Apex code as possible, including all the branches of conditional logic.
- Use the **System.runAs** method to test record level access for users in the code being tested.
- Always handle all exceptions that are caught, instead of merely catching the exception.
- Make use of `system.assert` and `system.assertEquals` statements in your Test Classes and methods to verify expected results.
- Try to cover both positive and negative test scenario by passing valid and invalid inputs to the Apex code.
- Use `startTest` and `stopTest` methods to avoid Governor Limits while testing actual code of functionality. All test data should be created outside and prior to calling these methods.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs



SALESFORCE HULK

CHAPTER - 10

GOVERNOR LIMITS AND BATCH APEX

WHAT IS A GOVERNOR LIMIT?

Governor limits are runtime limits enforced by the Apex runtime engine to ensure that code does not throw error. As Apex runs in a shared, multi tenant environment, the Apex runtime engine strictly enforces a number of limits to ensure that code does not monopolize shared resources. Resources such as CPU, processor, and bandwidth are shared by Apex on the Salesforce server.

Governor Limits:

1. Monitor and manage platform resources such as memory, database resources, etc.
2. Enable multi tenancy by ensuring the resources are available for all tenants on the platform.
3. Can issue program-terminating runtime exceptions when limit is exceeded.
4. Are typically reset per transaction.
5. Are subject to change from release to release.

Governor Limits basically cover:

1. Memory.
2. Database Resources.
3. Number of script statements.
4. Number of records processed.

TYPES OF GOVERNOR LIMITS:

Let's get a quick overview of different types of Governors limits available within the Salesforce.

1. **Per-Transaction Apex Limits:** Per-Transaction Apex Limits count for each Apex transaction.

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued	100	200
Total number of records retrieved by SOQL queries	50,000	
Total number of records retrieved by Database.getQueryLocator	10,000	
Total number of SOSL queries issued	20	

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com



training@cyntexa.com



[Salesforce Hulk](#)



Total number of records retrieved by a single SOSL query	2,000	
Total number of DML statements issued	150	
Total number of records processed as a result of DML statements, Approval.process, or database.emptyRecycleBin	10,000	
Total stack depth for any Apex invocation that recursively fires triggers due to insert, update, or delete statements	100	200
Total number of callouts (HTTP requests or Web services calls) in a transaction	50,000	
Maximum timeout for all callouts (HTTP requests or Web services calls) in a transaction	10,000	
Total number of sendEmail methods allowed	20	

For example:

```
for(Position pos : [SELECT Id, Name, FROM Position__c]){
    List<Job_Application__c> jobApp = [SELECT Id
                                     FROM Job_Application__c
                                     WHERE Position__c = pos.Id];
}
```

2. **Per-Transaction Certified Managed Package Limits:** Per-Transaction Certified Managed Package have passed the security review for AppExchange and get their own set of limits for most per-transaction limits.

Description	Cumulative Cross-Namespace Limit
Total number of SOQL queries issued	1,100
Total number of SOSL queries issued	220
Total number of DML statements issued	1,650

3. **Force.com Platform Apex Limits:** Force.com Platform Apex Limits aren't specific to an Apex transaction and are enforced by the Force.com platform.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com



training@cyntexa.com



[Salesforce Hulk](#)



Description	Limit
Maximum number of Apex classes scheduled concurrently	100
Maximum number of batch Apex jobs in the Apex flex queue that are in Holding status	100
Maximum number of batch Apex jobs queued or active concurrently	5
Maximum number of batch Apex job start method concurrent executions	1
Maximum number of batch jobs that can be submitted in a running test	5

4. **Static Apex Limits:** Let's get a quick overview of different types of static Apex limits available within the Salesforce.


Description	Limit
Default timeout of callouts (HTTP requests or Web services calls) in a transaction	10 seconds
Maximum SOQL query run time before Salesforce cancels the transaction	120 seconds
Maximum number of class and trigger code units in a deployment of Apex	5,000
For loop list batch size	200


5. **Size-Specific Apex Limits:** Size-Specific Apex Limits are the limits put on the length of code.

Description	Limit
Maximum number of characters for a class	1 million
Maximum number of characters for a trigger	1 million
Method size limit	65,535 bytecode instructions in compiled form
Maximum amount of code used by all Apex code in an	3 MB

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 


[Salesforce Hulk](#) 


organization	
--------------	--

6. **Miscellaneous Apex Limits:** Developers receive an error message when a non-selective query in a trigger executes against an object that contains more than 200,000 records and the maximum number of records that an event report returns for a user who is not a system administrator is 20,000. The limit is 100,000 for system administrators.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



WHAT IS BATCH APEX?

Batch Apex operates over small batches of records, covering your entire record set and breaking the processing down to manageable chunks. We could build a data cleansing operation that goes through all Accounts and Opportunities on a nightly basis and updates them if necessary, based on custom criteria.

Batch Apex is exposed as an interface that must be implemented by the developer. Batch jobs can be programmatically invoked at runtime using Apex.

NEED OF BATCH APEX:

As we know about the salesforce governor limits on its data. When you want to fetch thousands of records or fire DML on thousands of rows on objects it is very complex in salesforce and it does not allow you to operate on more than certain number of records which satisfies the Governor limits.

But for medium to large enterprises, it is essential to manage thousands of records every day. Adding/editing/deleting them when needed.

Salesforce has come up with a powerful concept called Batch Apex. Batch Apex allows you to handle more number of records and manipulate them by using a specific syntax.

We have to create a global apex class which extends Database.Batchable Interface because of which the salesforce compiler will know, this class incorporates batch jobs. Below is a sample class which is designed to delete all the records of Account object (Let's say your organization contains more than 50 thousand records and you want to mass delete all of them).

Methods:

Database.Batchable interface contains 3 methods that must be implemented:

1. `start()` :- It collects the records or objects to pass to the interface method `execute()`, call the `start()` at the beginning of a `BatchApexJob`. This method returns either a `Database.QueryLocator` object that contains the records passed to the job.
2. `execute()` :- To do the required processing of each chunk of data, use the `execute` method. This method is called for each batch of records that you pass to it. This method takes a reference to the `Database.BatchableContext` object.
3. `finish()` :- To send confirmation emails or execute post-processing operations, we use `finish()`. This method is called after all batches are processed.

Note: The order of execution of batches is not guaranteed.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



Example:

```
global class deleteAccounts implements Database.Batchable
{
    global final String Query;
    global deleteAccounts(String q)
    {
        Query=q;
    }

    global Database.QueryLocator start(Database.BatchableContext BC)
    {
        return Database.getQueryLocator(query);
    }


    global void execute(Database.BatchableContext BC, List scope)
    {
        List <Account> lstAccount = new list<Account>();
        for(Subject s : scope)
        {
            Account a = (Account)s;
            lstAccount.add(a);
        }
        Delete lstAccount;
    }

    // Send an email to the User after your batch completes
    global void finish(Database.BatchableContext BC)
    {
        Messaging.SingleEmailMessage mail = new
        Messaging.SingleEmailMessage();
        String[] toAddresses = new String[] {'sforce2009@gmail.com'};
        mail.setToAddresses(toAddresses);
        mail.setSubject('Apex Batch Job is done');
        mail.setPlainTextBody('The batch Apex job processed ');
        Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
    }
}

// This is how the batch class is called.
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



```
id batchinstanceid = database.executeBatch(new deleteAccounts('select Id from Account'));
```

SCHEDULE BATCH APEX:

Use the Apex scheduler and the Schedulable interface if you have specific Apex classes that you want to run on a regular basis, or to run a batch Apex job using the Salesforce user interface.

The scheduler runs as system—all classes are executed, whether or not the user has permission to execute the class.

Important: Salesforce schedules the class for execution at the specified time. Actual execution may be delayed based on service availability.

To schedule jobs using the Apex scheduler:

1. Implement the Schedulable interface in an Apex class that instantiates the class you want to run.
2. From Setup, enter Apex Classes in the Quick Find box, select Apex Classes, and then click Schedule Apex.
3. Specify the name of a class that you want to schedule.
4. Specify how often the Apex class is to run.
5. For Weekly—specify one or more days of the week the job is to run (such as Monday and Wednesday).
6. For Monthly—specify either the date the job is to run or the day (such as the second Saturday of every month.)
7. Specify the start and end dates for the Apex scheduled class. If you specify a single day, the job only runs once.
8. Specify a preferred start time. The exact time the job starts depends on service availability.
9. Click Save.

Note: You can only have 100 active or scheduled jobs concurrently.

Alternatively, you can call the `System.scheduleBatch` method to schedule the batch job to run once at a future time.

Once the job has been completed, you can see specifics about the job (such as whether it passed or failed, how long it took to process, the number of records process, and so on).

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE




Example:

```
global class ClassName implements Schedulable
{
    global void execute(SchedulableContext sc)
    {
        BatchClass b = new BatchClass(); // Your batch class
        database.executeBatch(b);
    }
}
```

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com 

training@cyntexa.com 

[Salesforce Hulk](#) 

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



S2 Labs



SALESFORCE HULK

CHAPTER - 11

VISUALFORCE



SALESFORCE DEVELOPMENT COURSE



WHAT IS VISUALFORCE?

Visualforce is a framework that allows developers to build sophisticated, custom user interfaces that can be hosted natively on the Lightning platform. The Visualforce framework includes a tag-based markup language, similar to HTML, and a set of server-side “standard controllers” that make basic database operations, such as queries and saves, very simple to perform.

In the Visualforce markup language, each Visualforce tag corresponds to a coarse or fine-grained user interface component, such as a section of a page, a related list, or a field. The behavior of Visualforce components can either be controlled by the same logic that is used in standard Salesforce pages, or developers can associate their own logic with a controller class written in Apex.

Sample of Visualforce Components and their Corresponding Tags

An Apex page with callouts to the `apex:page`, `apex:commandLink`, `apex:image`, `apex:relatedList`, `apex:pageBlock`, `apex:dataTable`, and `apex:detail` tags

What is a Visualforce Page?

Developers can use Visualforce to create a Visualforce page definition. A page definition consists of two primary elements:

- Visualforce markup
- A Visualforce controller

Visualforce Markup

Visualforce markup consists of Visualforce tags, HTML, JavaScript, or any other Web-enabled code embedded within a single `<apex:page>` tag. The markup defines the user interface components that should be included on the page, and the way they should appear.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



Visualforce Controllers

A Visualforce controller is a set of instructions that specify what happens when a user interacts with the components specified in associated Visualforce markup, such as when a user clicks a button or link. Controllers also provide access to the data that should be displayed in a page, and can modify component behavior.

A developer can either use a standard controller provided by the Lightning platform, or add custom controller logic with a class written in Apex:

A standard controller consists of the same functionality and logic that is used for a standard Salesforce page. For example, if you use the standard Accounts controller, clicking a Save button in a Visualforce page results in the same behavior as clicking Save on a standard Account edit page.

If you use a standard controller on a page and the user doesn't have access to the object, the page will display an insufficient privileges error message. You can avoid this by checking the user's accessibility for an object and displaying components appropriately.

A standard list controller enables you to create Visualforce pages that can display or act on a set of records. Examples of existing Salesforce pages that work with a set of records include list pages, related lists, and mass action pages.

A custom controller is a class written in Apex that implements all of a page's logic, without leveraging a standard controller. If you use a custom controller, you can define new navigation elements or behaviors, but you must also reimplement any functionality that was already provided in a standard controller.

Like other Apex classes, custom controllers execute entirely in system mode, in which the object and field-level permissions of the current user are ignored. You can specify whether a user can execute methods in a custom controller based on the user's profile.

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntaxa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.



SALESFORCE DEVELOPMENT COURSE



A controller extension is a class written in Apex that adds to or overrides behavior in a standard or custom controller. Extensions allow you to leverage the functionality of another controller while adding your own custom logic.

Because standard controllers execute in user mode, in which the permissions, field-level security, and sharing rules of the current user are enforced, extending a standard controller allows you to build a Visualforce page that respects user permissions. Although the extension class executes in system mode, the standard controller executes in user mode. As with custom controllers, you can specify whether a user can execute methods in a controller extension based on the user's profile.

Note :

Although custom controllers and controller extension classes execute in system mode and thereby ignore user permissions and field-level security, you can choose whether they respect a user's organization-wide defaults, role hierarchy, and sharing rules by using the with sharing keywords in the class definition. For information, see “Using the with sharing, without sharing, and inherited sharing Keywords” in the Apex Developer Guide.

Where Can Visualforce Pages Be Used?

Developers can use Visualforce pages to:

- Override standard buttons, such as the New button for accounts, or the Edit button for contacts
- Override tab overview pages, such as the Accounts tab home page
- Define custom tabs
- Embed components in detail page layouts
- Create dashboard components or custom help pages
- Customize, extend, or integrate the sidebars in the Salesforce console (custom console components)
- Add menu items, actions, and mobile cards in the Salesforce app

By: Shrey Sharma | Mobile: +91 75 6869 7474

ShreySharma.com

training@cyntexa.com

[Salesforce Hulk](#)

Disclaimer: All text, graphics and design are ©2018 Shrey Sharma. All rights reserved.